MODÉLISATION DE PROCÉDÉS LOGICIELS À BASE DE PATRONS RÉUTILISABLES

Hanh Nhi Tran

▶ **To cite this version:**

Hanh Nhi Tran. MODÉLISATION DE PROCÉDÉS LOGICIELS À BASE DE PATRONS RÉUTILISABLES. Génie logiciel [cs.SE]. Université Toulouse le Mirail - Toulouse II, 2007. Français. NNT : . tel-00545951

HAL Id: tel-00545951

https://theses.hal.science/tel-00545951

Submitted on 13 Dec 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

présentée

pour obtenir

LE TITRE DE DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

délivré par l'Université Toulouse II- Le Mirail

École doctorale : **MITT**

Spécialité : **Informatique**

par

HANH NHI TRAN

MODÉLISATION DE PROCÉDÉS LOGICIELS À BASE DE PATRONS RÉUTILISABLES

Soutenue le 8 novembre 2007 devant le jury composé de :

Marie-Pierre	**GERVAIS**	Rapporteur	Professeur à l'Université de Paris X
Jean-Pierre	**GIRAUDIN**	Rapporteur	Professeur à l'Université de Grenoble
Tuong Vinh	**HO**	Rapporteur	Directeur de recherche à l'IFI Hanoi
Nhan	**LE THANH**	Examinateur	Professeur à l'Université de Nice Sophia Antipolis
Bich Thuy	**DONG**	Co-Directrice	Professeur à l'Université Nationale du Vietnam à HoChiMinh
Bernard	**COULETTE**	Directeur	Professeur à l'Université de Toulouse II

à mes parents,

qui m'ont donné les ailes pour m'envoler vers mon rêve…

Remerciements

Ce mémoire de thèse représente l'achèvement de quatre années d'un travail laborieux mais enrichissant qui n'aurait pas pu voir le jour sans les conseils, l'aide, et la présence de nombreuses personnes.

Je tiens ainsi à remercier…

… madame **Dong thi Bich Thuy**, *Professeur à l'Université Nationale du VietNam à HoChiMinh, ma co-directrice de thèse, qui a été la première à m'encourager à poursuivre mes études et m'a soutenue pendant mes années de doctorat. Elle a également été l'inspiratrice de mon envie de devenir une bonne enseignante. Bien sûr, ce qu'elle m'a apporté ne se limite pas au seul domaine scientifique ou pédagogique et je lui en serai longtemps reconnaissante.*

… monsieur **Bernard Coulette**, *Professeur à l'Université de Toulouse le Mirail, mon directeur de thèse, pour m'avoir accueillie dans son équipe et m'avoir guidée du premier au dernier jour de thèse, toujours avec beaucoup de patience et de disponibilité. Sans ses conseils, ses remarques et surtout ses relectures, ses corrections minutieuses du mémoire, je ne serais certainement pas là où j'en suis. Je n'oublierai pas non plus les attentions touchantes de sa famille pendant mon séjour en France ; je voudrais exprimer ici ma gratitude envers eux.*

… madame **Marie-Pierre Gervais**, *Professeur à l'Université de Paris X, Monsieur* **Jean-Pierre Giraudin**, *Professeur à l'Université de Grenoble, et Monsieur* **Ho Tuong Vinh**, *Professeur à l'IFI de Hanoi, pour m'avoir fait l'honneur de rapporter et juger mes travaux.*

… monsieur **Khan Le Thanh**, *Professeur à l'Université de Nice Sophia Antipolis, pour avoir accepté de participer à mon jury de thèse.*

… l'équipe **Isycom** *de l'UTT et l'équipe* **Macao** *de l'IRIT, mes équipes d'accueil pendant quatre ans, pour m'avoir acceptée et offert d'excellentes conditions de travail. Merci en particulier à Rédouane, Xavier, Sophie, Pierre-Jean, Taoufiq, non seulement pour leurs commentaires enrichissants lors de mes présentations orales, mais aussi pour leurs conseils pratiques qui m'ont permis de régler les petits soucis du quotidien. Merci également à Jean-Christophe et Cathy pour leurs aides durant mon année d'ATER à l'UTT.*

… **mes collègues au département de Technologie de l'Information** *à l'Université Nationale du VietNam à HoChiMinh, pour m'avoir soutenue tout au long de ces années. Je voudrais remercier tout particulièrement Thay Tran Dan Thu, pour l'origine de cette thèse, et pour plusieurs longues discussions qui m'ont beaucoup apporté sur le plan scientifique et méthodologique. Merci spécialement*

à Thay Nguyen Dinh Thuc, Anh Duong Anh Duc pour m'avoir toujours fait confiance et de m'avoir encouragée dans les moments obscurs qui ont parsemé ces quatre années. Merci à Co Bich, Duy, Minh Son, Vu, Duan, Thuy Anh, Dong pour leurs mails et chats qui ont été une agréable touche d'humour ayant indéniablement réduit ma solitude. Merci à Minh Tuan pour sa contribution à la réalisation du prototype illustrant mon travail de thèse.

*. . . **mes amis et mes étudiants Vietnamiens à Toulouse**, Co Anh, Chi Dung, Co Ngoc, Chu The, Anh Ti, Chi Loc, Chi Hai Van, Dinh, Xuan Dung, Tran, Hau, Yen et les autres, pour leur amitié sincère pendant mes moments de nostalgie. Je voudrais exprimer ma profonde gratitude à Co Anh pour ses attentions délicates et ses soins touchants durant ces années. Un grand merci à Chi Dung pour avoir consacré beaucoup de temps et d'efforts pour préparer mon pot de thèse. Tous mes chers compatriotes vont me manquer (et leurs repas festifs) !*

*. . .**mes amis de l'équipe de recherche**, Mohamed, Yaël, Nicolas, Adil, Younes et Benoît, pour avoir rendu mon travail plus amusant et m'avoir soutenue lors des périodes difficiles. Merci en particulier à Yaël, qui a partagé avec moi non seulement le bureau, mais aussi son impressionnante collection de musique ainsi que ses nombreuses connaissances. Je n'oublie pas Mahmoud, Salima, et Jean-François pour les séances de cinéma et les très agréables discussions au café au début de mon séjour. Ces belles années toulousaines resteront pour moi des souvenirs inoubliables.*

*. . .**mes amis d'enfance**, Quynh Anh, Hue, Nga, Tung, Hieu, Cao Son, Huong, Uyen et les autres, pour avoir gardé le contact avec moi toutes ces années et pour être la source de mes meilleurs souvenirs. Un gros merci tout particulièrement à Minh Hong ; j'apprécie vraiment notre vieille complicité.*

*Mes derniers remerciements sont pour **ma chère famille**, pour son soutien tout au long de mon existence. Merci à Maman et Papa pour leur amour inconditionnel et sans limite. Merci à Chi Linh Anh Tri, Anh Phong Chi Ha, Khue Quang et Nam Tram, qui chacun à sa manière, par ses paroles et ses gestes, m'aident et m'encouragent depuis toujours. Merci à mes beaux-parents pour leur compréhension et leur soutien. Enfin, un énorme merci à Anh Hung, mon mari, et à ma petite Song Nhi pour leur doux amour et leur grand sacrifice durant ces longues années. Je leur dois beaucoup et les aime de tout mon coeur.*

TABLE DES MATIÈRES

TABLE DES FIGURES

INTRODUCTION GÉNÉRALE

Contexte

Le travail présenté dans cette thèse s'inscrit dans le domaine de *l'ingénierie des procédés*[1] *logiciels* (Software Process Engineering), une discipline du Génie Logiciel qui vise la maîtrise du développement de logiciels en mettant l'accent sur le support et le contrôle du processus de développement.

En s'appuyant sur le constat reconnu dès les années 1980 que la qualité du processus de développement conditionne largement la qualité du produit élaboré [Humphrey88] [Armenise93], l'ingénierie des procédés développe des méthodes et des techniques pour modéliser, assister, évaluer et améliorer les processus de développement. On peut dire que l'ingénierie des procédés se consacre au développement d'un produit particulier : le procédé de développement.

Comme pour le développement du logiciel, le double défi auquel est confrontée l'ingénierie des procédés est celui de la qualité et de la productivité. Cependant, pour l'ingénierie des procédés logiciels, ces défis sont particulièrement difficiles à relever car les procédés de développement de logiciel sont des produits intrinsèquement complexes, pouvant être mis en oeuvre de manière répartie, coopérative, itérative, par différents types d'agents, avec différentes contraintes de performance, et nécessitant diverses ressources matérielles ou humaines. Comme pour le développement du logiciel, la réutilisation est considérée comme un moyen efficace pour surmonter ces défis. A cet effet, plusieurs travaux de recherche ont été menés dans le domaine. Cependant, les résultats obtenus sont encore limités, en particulier ce qui concerne la réutilisation au cours de la modélisation de procédés. La plupart des travaux sont centrés sur la capitalisation d'expériences de construction de procédés qui sont généralement représentées de façon informelle et dont la réutilisation est faite de façon manuelle.

C'est dans ce contexte que nous nous intéressons au problème de la réutilisation dans la modélisation de procédés logiciels. L'origine de cette thèse est le projet RHODES [Coulette00] [Crégut97] développé à l'IRIT-ENSEEIHT, qui a donné lieu à la réalisation d'un langage de description de procédés exécutable, et d'un AGL centré procédé. Les expérimentations réalisées sous RHODES ont montré qu'il était important de pouvoir capitaliser l'expérience acquise lors de la définition d'un procédé, et qu'il fallait offrir aux concepteurs de procédés des moyens pour réutiliser cette expérience. Cette idée a été développée dans la thèse de Tran D.T. [TranDT01] qui a étudié la formalisation du concept de composant de procédé[Coulette01][TranDT05].

[1] En fait, le terme «process» en anglais peut être traduit en «procédé» ou «processus» en français. Nous distinguons intentionnellement ces deux concepts : un procédé est une description (statique) d'une démarche de développement alors qu'un processus est l'exécution (dynamique) de cette démarche. Autrement dit, un processus est l'exécution d'un procédé.

Dans ce travail, Tran a abordé la notion de patron de procédé comme cas particulier d'un composant de procédé, mais le concept de patron de procédé en tant que tel n'a pas été formalisé.

C'est pour cette raison que nous avons décidé d'étudier la réutilisation de procédés logiciels selon une approche à base de patrons. Cette thèse, commencée au sein de l'équipe d'accueil GRIMM puis terminée au sein du laboratoire IRIT, s'inscrit dans le contexte d'une collaboration en recherche entre l'Université de Toulouse II et l'Université Nationale du Vietnam à HoChiMinh Ville. Elle a été réalisée dans le cadre d'une convention de cotutelle entre ces universités.

Motivations

Inspiré par le succès des patrons dans le domaine du développement du logiciel, Coplien [Coplien94] a introduit la notion de *patron de procédé* au sens d'un patron qui capture des connaissances de procédé et qui peut être réutilisé ultérieurement pour faciliter la construction d'autres procédés. Depuis, plusieurs travaux dans les domaines des procédés logiciels, des procédés métier (Business Processes), des workflows et de l'ingénierie de méthodes ont employé cette notion.

Pourtant, en étudiant les propositions existantes pour représenter et réutiliser les patrons de procédé, nous avons fait le constat d'un certain nombre de lacunes, que l'on peut synthétiser comme suit :

- **Définition vague et limitée du concept de patron de procédé**

Il existe indéniablement une confusion terminologique et conceptuelle par rapport au concept de patron de procédé. D'une part, le terme de «patron de procédé» n'est pas utilisé avec la même signification par tous les auteurs, d'autre part peu de définitions précises de cette notion de patron de procédé ont été données.

La plupart des travaux existants considèrent les patrons de procédé comme des entités capturant des activités de développement réutilisables sous forme de modules de base pour construire de nouveaux procédés [Ambler98][Bergner98][Dittmann02][Hagen04]. Cette manière d'appréhender les patrons de procédé est réductrice et ne couvre pas la diversité des connaissances sur les procédés. Nous pensons que les patrons de procédé peuvent capturer aussi d'autres connaissances plus abstraites (par exemple des structures communes, des conceptions génériques de procédés [Storrle01][Iida02]), et peuvent servir à d'autres besoins de modélisation, par exemple pour restructurer ou enrichir les procédés.

- **Formalisation insuffisante de la notion de patron de procédé**

La plupart des patrons de procédé proposés dans la littérature sont représentés informellement. Par conséquent, leur réutilisation dans la modélisation de procédés est difficile et non automatisable. Pour permettre la modélisation de procédés à base des patrons réutilisables, il faut disposer de Langages de Description de Procédés (LDP) supportant la représentation de patrons et leur mise en œuvre (application). Il existe encore peu de tels langages. À notre connaissance, le concept de patron de procédé n'est défini explicitement que

dans le framework *Living Software Development Process* [Gnatz03], le langage de description de patrons *PROPEL* [Hagen04], le langage de description de procédés *PROMENADE* [Ribo02] et le méta-modèle de procédé logiciel *SPEM* [SPEM07]. *Living Software Development Process* permet une description flexible de procédé mais ne propose pas une définition détaillée du concept de patron de procédé. PROPEL décrit la structure interne des patrons ainsi que les relations entre patrons, mais ne traite pas leur réutilisation. PROMENADE fournit un LDP formel et supporte les patrons de procédé génériques en utilisant la paramétrisation, mais ignore l'organisation des patrons. SPEM a l'avantage d'être le méta-modèle standardisé par l'OMG et dédié aux procédés logiciels. Cependant, sa version actuelle (SPEM 1.0) n'intègre pas le concept de patron de procédé, et la version en cours d'adoption (SPEM 2.0) ne met pas spécialement l'accent sur le concept de patron de procédé, mais fournit des concepts et des mécanismes généraux pour réutiliser les procédés en séparant les méthodes et leurs applications dans les procédés.

Aucun des travaux cités ci-dessus ne fournit une formalisation suffisance de la notion de patron de procédé. Ils se contentent de formaliser des patrons capturant des activités de développement réutilisables, sans prendre en compte d'autres types de patrons reflétant des connaissances plus abstraites sur les procédés. De plus, ces travaux réduisent le plus souvent l'application des patrons de procédé à une réutilisation assez simpliste de modules de base pour construire de nouveaux procédés.

- **Réutilisation manuelle de patrons de procédé**

La réutilisation (application) de patrons de procédé existants est souvent faite de façon manuelle et informelle. D'une part, cela ne permet pas de réduire les efforts de modélisation de procédés, d'autre part cela peut conduire à des applications incorrectes. Sur le plan méthodologique, il n'existe pas de méthode bien définie pour guider l'application systématique de patrons au cours de la modélisation. S'il existe des guides méthodologiques utilisés dans certains projets, ceux-ci restent informels et ne sont pas intégrés dans un méta-procédé automatisable. Le manque d'outils support et de méthodes rigoureuses guidant les concepteurs dans l'exploitation des patrons de procédé est aussi un facteur limitant pour leur réutilisation.

Compte tenu de ces limites, et comme le montrera l'étude plus détaillée sur les travaux existants dans la suite de ce document, les questions suivantes nous semblent pertinentes : Qu'est-ce qu'un patron de procédé ? Comment représenter un patron et la façon de le réutiliser ? Comment appliquer efficacement des patrons de procédé dans la modélisation de procédés ? Comment aider les concepteurs de procédés à modéliser un procédé en réutilisant des patrons ?

Notre ambition dans cette thèse est d'apporter des réponses convaincantes à ces questions.

Objectifs et Approches

L'objectif général de notre travail est de mettre en œuvre efficacement la représentation et la réutilisation de patrons de procédé dans la modélisation de procédés. Pour cela, notre travail est décliné selon deux axes principaux :

- **Définition d'un Langage de Description de Procédés (LDP) intégrant le concept de patron de procédé**

Notre premier objectif est de fournir une définition rigoureuse du concept de patron de procédé, suffisamment générale pour couvrir différents types de connaissances sur les procédés, y compris la manière d'appliquer les patrons. Une fois le concept de patron de procédé défini, il s'agit de proposer un LDP intégrant ce concept afin de permettre la représentation de procédés à base de patrons réutilisables. L'exécutabilité de ce LDP est naturellement souhaitable, mais son obtention sort du cadre de notre travail.

Nous nous sommes fixés les objectifs suivants pour la définition d'un tel LDP :

(1) Le LDP devra prendre en compte les langages de modélisation existants, en particulier ceux qui sont standardisés, pour faciliter sa diffusion et son alignement avec les autres LDP.

(2) Le LDP devra être suffisamment formel pour faciliter son support par des outils.

(3) Le LDP devra être aussi simple et compréhensible que possible, pour permettre aux concepteurs de modéliser les procédés aisément et rapidement.

Nous avons donc décidé de définir notre LDP par l'intermédiaire d'un méta-modèle (appelé **UML-PP**) conforme au MOF [MOF06]. Ce choix nous permet d'une part de définir formellement notre LDP en réutilisant l'infrastructure d'UML 2.0 [UML05a] qui est le standard de facto en modélisation du logiciel, d'autre part de pouvoir profiter des outils existants qui supportent MOF et UML. Nous pouvons donc satisfaire l'objectif (2). Pour satisfaire l'objectif (1), nous nous somme inspirés de SPEM 1.1 [SPEM05], la version actuelle du méta-modèle de l'OMG dédié à la description des procédés logiciels. Pour ce qui concerne la simplicité de UML-PP (objectif (3)), nous avons décidé ne pas le définir comme un profil SPEM ou un profil UML, mais par un méta-modèle conforme au MOF qui ne réutilise qu'un noyau d'UML et les concepts de SPEM nécessaires pour décrire les procédés. Le choix s'est porté sur la version 1.1 de SPEM car SPEM 2.0 [SPEM07] — au demeurant très complexe — n'est pas encore standardisé.

- **Définition d'une méthode de modélisation de procédés à base de patrons réutilisables**

Notre second objectif est de fournir des moyens facilitant la réutilisation de patrons de procédé en modélisant les procédés. Ces moyens doivent d'une part permettre d'automatiser les applications de procédé pour réduire les efforts de modélisation, d'autre part aider les concepteurs à réutiliser systématiquement et efficacement les patrons de procédé pour modéliser les procédés.

Pour ce faire, nous avons opté pour une méthode comportant des opérateurs de réutilisation des patrons de procédé, et une démarche de modélisation de procédés à base de patrons. Nous avons défini une sémantique opérationelle pour ces opérateurs pour permettre leur automatisation. Quant à la démarche de modélisation, nous l'avons décrite sous forme d'un méta-procédé à grain fin (appelé **PATPRO**) en utilisant notre LDP (défini par le méta-modèle) UML-PP.

Enfin, nous avons décidé de concrétiser les propositions ci-dessus en fournissant un environnement de modélisation supportant la réutilisation de patrons de procédé. Cet environnement (nommé **PATPRO-MOD**) offre aux concepteurs un éditeur graphique pour créer les modèles de procédé en UML-PP, en particulier les modèles basés sur les patrons. Il leur offre aussi des fonctions pour sélectionner des patrons à appliquer et des fonctions pour imiter automatiquement des patrons. La démarche de modélisation prescrite dans cet environnement est guidée par le méta-procédé PATPRO.

Plan de la thèse

Outre cette introduction, la thèse est organisée en quatre chapitres.

Dans le premier chapitre, consacré à l'**état de l'art**, nous présentons une étude bibliographique sur la modélisation de procédés, la réutilisation de procédés et les patrons de procédé.

Le second chapitre porte sur la **formalisation du concept de patron de procédé**. Nous présentons d'abord notre définition du concept de patron de procédé et de leurs applications. Ces propositions sont ensuite formellement définies à travers le méta-modèle UML-PP.

Le troisième chapitre est dédié au développement d'une **méthode de modélisation de procédés à base de patrons réutilisables**. Nous définissons les opérateurs de réutilisation de patrons de procédé, et proposons le méta-procédé PATPRO qui préconise une démarche de modélisation de procédé à base de patrons.

Dans le quatrième chapitre, nous présentons la réalisation de l'environnement PATPRO-MOD qui fournit **un outil de modélisation de procédés supportant la réutilisation de patrons**. Une étude de cas est également présentée pour valider notre approche et illustrer l'utilisation de cet outil.

La **conclusion** synthétise les contributions de cette thèse. Elle met en évidence également les diverses perspectives de ce travail.

Le document est conclu par trois annexes : l'**annexe A** décrit les algorithmes définissant la sémantique opérationnelle des opérateurs de réutilisation de patrons de procédé ; l'a**nnexe B** décrit le procédé VUP (*View-based Unified Process*) utilisé dans l'étude de cas du chapitre IV ; l'a**nnexe C** montre un extrait du code C# de l'environnement PATPRO-MOD.

CHAPITRE I.

ÉTAT DE L'ART

Notre travail repose sur l'hypothèse que le concept de patron de procédé peut être employé pour faciliter la modélisation de procédés en favorisant la réutilisation de connaissances de procédé. Le but de ce chapitre est de faire le point sur les travaux existants relatifs à cette problématique.

Nous présentons d'abord dans une première section le domaine de notre travail : la modélisation de procédé. L'objectif n'est pas de fournir un état de l'art complet sur la modélisation de procédés, mais de faire une synthèse des concepts de base du domaine, des besoins attendus pour la modélisation de procédés, et des principales approches de modélisation de procédés.

Dans un second temps, nous discutons de la réutilisation de procédés : sa problématique et les principales solutions existantes. Sur la base de cette discussion, nous proposons un cadre de référence pour caractériser les différentes approches de réutilisation de procédés.

La troisième section est consacrée aux patrons de procédé. Premièrement, nous introduisons la notion de patron de procédé. Ensuite, nous présentons des travaux sur la modélisation de procédés qui supportent le concept de patron de procédé. Cette présentation est suivie par une évaluation des travaux décrits en utilisant le cadre de référence proposé.

Dans la conclusion, nous analysons les acquis et les limites des approches existantes pour réutiliser les procédés par l'intermédiaire de patrons de procédé. Cela nous permet de justifier notre proposition décrite dans la suite de la thèse.

I. MODÉLISATION DE PROCÉDÉS

Un *procédé logiciel* (software process) est, en général, défini comme un ensemble d'activités aussi bien techniques qu'administratives pour développer et pour maintenir un produit logiciel [Feiler93][Lonchamp93]. Même si elles exhibent des niveaux différents de sophistication dans la maîtrise de leur procédé, toutes les organisations développant du logiciel suivent un procédé.

De nos jours, il est acquis que la qualité des procédés logiciels conditionne la qualité du produit à réaliser [Humphrey88][Armenise93]. D'où l'importance de *l'ingénierie des procédés* (Software Process Engineering), une discipline du génie logiciel dont l'objectif est de supporter les procédés logiciels en fournissant les moyens de modéliser, d'analyser, d'améliorer, de mesurer et d'automatiser les activités de développement [Finkelstein94][Derniame99].

Pour pouvoir analyser, améliorer, mesurer et automatiser les procédés logiciels, il faut d'abord les définir explicitement. C'est l'objectif de l'axe de recherche principal de l'ingénierie des procédés, la *modélisation de procédés logiciels* (Software Process Modeling).

La modélisation d'un procédé logiciel est l'élaboration d'une description de ce procédé en utilisant un (ou plusieurs) *modèle(s) de procédé* (process model) [Feiler93]. Dans cette définition, un modèle de procédé est une abstraction de procédé représenté par un *Langage de Description de Procédés* (Process Modeling Language).

Pendant ces deux dernières décennies, de nombreux travaux se sont intéressés à la modélisation de procédés et les aspects qui lui sont liés. Notre l'intention ici n'est pas de fournir une étude exhaustive de l'état de l'art du domaine mais de présenter une vue d'ensemble de la modélisation de procédés. Plus précisément, nous récapitulons dans la suite de ce chapitre les objectifs de la modélisation de procédés, les perspectives de modélisation, les approches de description de procédés et des méthodes de modélisation de procédés.

I.1. OBJECTIFS ET APPORTS DE LA MODÉLISATION

L'objectif principal de la modélisation de procédés est d'expliciter les pratiques de développement pour pouvoir les étudier, les améliorer et les utiliser de manière répétable, gérable et éventuellement automatisable [Finkelstein94].

Plus spécifiquement, on attribue généralement trois finalités à la modélisation de procédés [Rolland98] : *descriptive*, *prescriptive* et *explicative*. La modélisation descriptive décrit un procédé tel qu'il est (*as-is process*) dans le but de le comprendre et l'évaluer. La modélisation prescriptive définit un procédé souhaité en décrivant comment il devra être (*to-be process*). La finalité prescriptive vise à guider les intervenants et à assister l'exécution de procédé par l'intermédiaire d'outils. La finalité explicative vise à modéliser un procédé en fournissant la logique du procédé.

Les différents travaux du domaine des procédés [Curtis92][Derniame99] s'accordent sur les avantages de la représentation explicite de procédé :

- *faciliter la compréhension des procédés* : en représentant explicitement un procédé, les connaissances de procédé sont concrétisées et deviennent compréhensives, communicables. Un modèle de procédé est donc un bon moyen pour faciliter la communication entre les intervenants d'un procédé.
- faciliter la gestion et l'exécution de procédé : si un modèle de procédé est défini de manière assez détaillée, et avec une sémantique claire, on peut développer des outils d'assistance pour la planification, le contrôle et la surveillance de procédé en suivant le modèle du procédé. Si le modèle est décrit formellement, on peut même l'exécuter automatiquement. C'est l'objectif des Ateliers de Génie Logiciel centrés-Procédé (AGL-P).
- faciliter l'analyse et l'amélioration de procédés : un modèle de procédé fournit l'information pour analyser et simuler le procédé. De plus, l'existence d'un modèle de procédé permet la réutilisation de procédés bien définis, ce qui est important pour leur amélioration.

Les caractéristiques requises d'un modèle de procédé dépendent bien entendu de la ou des finalités pour lesquelles il est développé.

I.2. MODÈLE DE PROCÉDÉ

Un modèle de procédé est une abstraction d'un procédé réel. Alors que la plupart des travaux du domaine s'accordent sur cette définition générale du modèle de procédé, il y a une divergence au sujet des éléments à décrire dans les modèles de procédé. Malgré l'absence d'un consensus global, nous pouvons mettre en évidence les points communs sur le contenu et les perspectives des modèles de procédé dans différentes études sur la modélisation de procédés [Dowson91][Feiler93][Lonchamp93][Armenise93][Finkelstein94][ACuna01].

Contenu du modèle de procédé

Divers éléments de procédé peuvent être représentés dans un modèle de procédé [Huff96]. En synthétisant les approches préconisées dans différents travaux [Benali92][Finkelstein94][Fugetta96][Derniame99][ACuna01][SPEM05], nous proposons la classification d'éléments de procédé suivante :

- ***Éléments primaires*** : ce sont les éléments principaux qui reflètent l'essence des procédés sans prendre en compte les aspects planification et exécution de procédé. Ce sont :
  - les *activités* : une activité est un ensemble d'actions à réaliser pour accomplir un objectif de développement[1].
  - les *produits* : un produit est un artéfact utilisé ou élaboré par les activités durant le développement.
  - les *rôles* : un rôle est un concept abstrait regroupant un ensemble de responsabilités et de compétences nécessaires pour réaliser certaines activités de développement.
- ***Éléments secondaires*** : ce sont les éléments qui fournissent des informations supplémentaires pour mettre en œuvre un procédé. Ce sont :

[1] Une *tâche* (Task) est parfois synonyme d'activité. Cependant, certains auteurs distinguent les deux termes : une tâche est une activité contrôlée, alors qu'une activité n'est pas contrôlée.

- les *agents* : un agent est une personne participant au développement. Un agent n'est pas un rôle, il peut jouer plusieurs rôles en même temps. Dans le sens inverse, un rôle n'est pas forcement associé à un agent, mais peut être associé à un groupe d'agents. Le concept d'agent est nécessaire pour la gestion de processus.
- les *ressources* : une ressource est un élément qui facilite l'exécution d'une activité, par exemple un outil, un guidage, etc. De telles informations sont utiles pour l'assistance au cours d'un processus.
- les *informations qualitatives* : elles permettent d'évaluer la performance et la qualité de procédés, par exemple des métriques, les résultats de révision ou test, etc.
- les *informations organisationnelles* : elles facilitent l'exécution d'un procédé pour un projet spécifique. Généralement, elles concernent le contexte de travail, l'organisation de ressources, de coopérations et de communications.

Les modèles de procédé ne décrivent pas seulement des éléments de procédé, mais aussi des relations entre eux. Les relations principales sont les relations entre activités et produits, entre rôles et activités et entre rôles et produits. La Figure I-1 montre les éléments de procédé et les relations les plus importants.

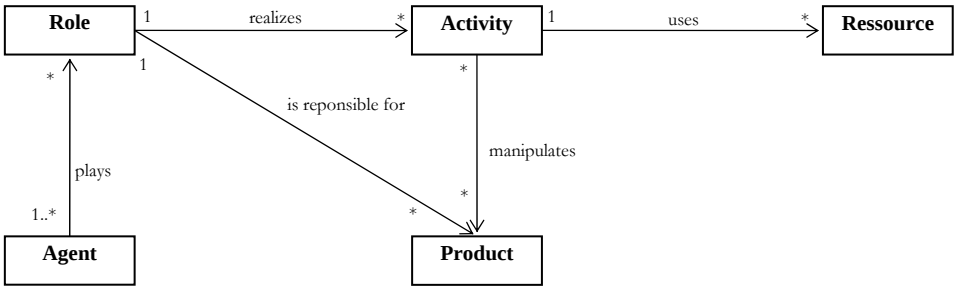


Figure I-1. Modèle conceptuel de procédés (adapté de [Derniame99])

Perspectives de modélisation

Un modèle de procédé peut être représenté sous plusieurs points de vue pour prendre en compte divers aspects du procédé. Il peut donc comprendre plusieurs sous-modèles reflétant différentes vues ou perspectives sur le procédé. Curtis et al. [Curtis92] ont proposé quatre perspectives de modélisation :

- *Vue fonctionnelle* représentant les activités à réaliser du procédé.
- *Vue comportementale* décrivant les différentes possibilités d'ordonner des activités et les conditions de leur enchaînement.
- *Vue organisationnelle* représentant les rôles qui exécutent les activités et la distribution de ressources pour l'exécution de procédé.
- *Vue informationnelle* représentant les informations manipulées et élaborées par le procédé, par exemple des données ou des produits.

Le choix de perspectives pour modéliser un procédé dépend des besoins de manipulation du procédé. Le contenu d'un modèle varie selon la perspective adoptée.

I.3. FORMALISME DE MODÉLISATION DE PROCÉDÉS

Un modèle de procédé doit être représenté en un certain langage. En fait, les premiers efforts de modélisation de procédés avec des langages informels sont assez anciens, comme par exemples les modèles de cycle de vie proposés dans [Royce70] et [Boehm88]. Mais à partir de la proposition de Osterweil [Osterweil87] sur la représentation explicite des procédés par des langages *exécutables*, la notion de *Langage de Description de Procédés* (LDP) a commencé à émerger. Depuis, deux générations de LDP[1] ont été développées [Conradi97] [Sutton97][ACuna01].

Un Langage de Description de Procédés est un formalisme de modélisation développé ou adapté pour décrire les procédés. Il définit les concepts dédiés aux procédés, et fournit une syntaxe et un système de notations pour représenter des modèles de procédé en utilisant ces concepts. Dans la suite, nous présentons les propriétés attendues d'un LDP et une classification des LDP existants.

I.3.1. Propriétés d'un Langage de Description de Procédés

Les propriétés attendues d'un LDP sont discutées dans plusieurs études [Armenise93][Ambriola97][Derniame99]. Nous présentons ci-après celles qui nous semblent les plus importantes :

- **Formalisation** : cette propriété reflète le niveau de rigueur de la définition de la syntaxe et de la sémantique d'un LDP. Il y a trois degrés de formalisation (dans l'ordre croissant) : informel, semi-formel et formel. Un langage informel est intuitivement défini, un langage semi-formel dispose d'une syntaxe formellement définie, et un langage formel dispose d'une syntaxe ainsi que d'une sémantique formellement définies. L'intérêt d'un LDP ayant un degré de formalisation élevé est qu'il permet une représentation précise des procédés et donc supporte plus facilement l'analyse et la vérification de modèles.
- **Expressivité** : l'expressivité d'un LDP reflète sa capacité à représenter les procédés. Pour avoir une riche expressivité, un LDP doit fournir tous les concepts nécessaires pour exprimer directement les éléments de procédé et selon différentes perspectives.
- **Compréhensibilité** : il y a deux facteurs qui influencent la compréhensibilité d'un LDP : sa notation et son degré de standardisation. Un LDP peut proposer des notations textuelles ou graphiques pour représenter les concepts qu'il définit. Au niveau de la notation, on aura tendance à privilégier les solutions graphiques dont les qualités sont indéniables ; la standardisation, quant à elle, favorise la compréhensibilité dans la mesure où elle favorise le partage de concepts et de notations communs par une communauté.
- **Abstraction et Modularité** : il est souhaitable qu'un LDP propose des mécanismes d'abstraction et d'agrégation pour permettre de structurer les procédés. Un LDP supportant l'abstraction et la modularité facilite la réutilisation de procédés.

[1] On considère que les LDP développés avant 1996 appartiennent à la première génération et ceux datés après 1996 constituent la deuxième génération.

- **Exécutabilité** : pour être exécutable, un LDP doit permettre de représenter les modèles avec une sémantique opérationnelle.
- **Évolutivité** : c'est la capacité de supporter l'évolution de modèles de procédé. La réflexion est une caractéristique importante d'un LDP pour atteindre l'évolutivité.

Il n'est pas facile pour un LDP de satisfaire tous ces besoins, car il y a souvent des conflits entre eux. Par exemple, il peut être contradictoire d'exiger qu'un LDP soit à la fois compréhensible et exécutable [Kellner91]. Par conséquent, dans la pratique, un LDP ne peut souvent viser qu'un sous-ensemble des propriétés ci-dessus.

I.3.2. Approches de définition de langages de description de procédés

Il y a différentes classifications des approches de définition de LDP, comme par exemple celles basées sur les paradigmes du langage [Armenise93][Ambriola97], sur le degré de formalisation du langage [Huff96][Zamli01] ou sur la finalité du langage [ACuna01].

Dans cet état de l'art, nous choisissons l'aspect formalisation pour catégoriser les approches de définition de LDP. Comme nous l'avons expliqué dans la section I.3.1, le degré de formalisation d'un LDP est mesuré par la rigueur de sa définition.

Les LDP peuvent être classifiés en trois catégories : informels, semi-formels et formels.

Langages informels

Un LDP informel est un langage qui n'est pas défini explicitement et formellement. Plus précisément, les LDP informels emploient certaines notations pour représenter les procédés, mais ils ne sont pas définis avec des concepts, une syntaxe et une sémantique qui donnent les règles de construction de modèles de procédé. Par conséquent, les modèles décrits par des LDP informels peuvent être imprécis, ambigus. Ils sont donc difficiles à analyser et non exécutables.

Les notations utilisées peuvent être textuelles (par exemple, langage naturel) ou graphiques (par exemple, les diagrammes de PERT ou de Gantt).

Au début du génie logiciel, les langages informels ont été largement utilisés pour décrire des modèles de cycle de vie, comme par exemple dans [Royce70][Boehm88]. Bien qu'imprécis, les langages informels sont néanmoins expressifs et donc encore employés pour décrire les procédés complexes, comme les modèles de procédé proposés par l'approche CMMI [CMMI02] et la norme ISO 12207 [ISO95].

Langages semi-formels

Un LDP semi-formel est un langage de modélisation de procédés défini syntaxiquement au niveau conceptuel. Un tel langage propose une définition explicite des concepts de procédé, une syntaxe formelle pour élaborer des modèles en utilisant les concepts proposés. La sémantique des concepts est définie, mais elle peut rester imprécise, incomplète et non opérationnelle. Les langages semi-formels peuvent supporter l'analyse, mais pas la simulation ou l'exécution de modèles. Les LDP semi-formels sont généralement utilisés pour spécifier et concevoir des procédés.

Le Tableau I-1 montre quelques LDP classés dans cette catégorie[1].

Année	LDP	AGL-P	Référence
1993	IDEF0 (SADT)	-	[IDEF093]
1994	E3	-	[Baldi94]
1994	Limbo	Oikos	[Montangero94]
1994	Base Model	PADM	[Bruynooghe94]
2001	SPEM	-	[SPEM05]

Tableau I-1. Exemples de LDP semi-formels

Langages formels

Comme les LDP semi-formels, les LDP formels sont des langages de modélisation de définis au niveau conceptuel, mais avec une définition plus rigoureuse. C'est-à-dire qu'un tel LDP définit les concepts de procédé, une syntaxe formelle pour décrire des modèles de procédé, et une sémantique opérationnelle pour manipuler ces modèles.

Les LDP formels sont vérifiables et simulables/exécutables. Ils sont utilisés pour élaborer des modèles prescriptifs dans le but d'assister les procédés décrits.

En général, un LDP formel s'inspire d'un paradigme de langage de programmation ou de modélisation de logiciels. Dans [Armenise93] les paradigmes suivants sont distingués :

- **Approche procédurale :** L'idée fondamentale est de représenter le modèle de procédé sous la forme d'un programme. Ce programme décrit de manière détaillée comment le procédé logiciel doit être réalisé.
- **Approche déclarative** : L'approche déclarative utilise des déclarations logiques (règles) pour décrire les procédés. Ceux-ci sont décrits en termes de résultats attendus par l'utilisateur sans détailler la manière dont ces résultats sont obtenus (de façon algorithmique).
- **Approche fonctionnelle** : L'approche fonctionnelle définit le procédé logiciel à travers un ensemble de fonctions mathématiques. Chaque fonction est décrite en termes de relations entre les données d'entrée et les données de sortie.
- **Approche basée sur les graphes ou les réseaux transitionnels** : Cette approche décrit le procédé logiciel à travers un réseau ou un graphe qui compose des nœuds et des arcs. Les réseaux de Petri sont souvent utilisés dans cette approche. Des nœuds représentent des activités, des arcs représentent les transitions entre les activités du procédé.
- **Approche basée sur UML**: Cette approche utilise des diagrammes d'UML pour représenter les concepts de procédé et renforce la sémantique de ces diagrammes avec un

[1] La colonne AGL - P montre l'éventuel environnement support

autre langage formel pour rendre les modèles de procédé à la fois compréhensifs et exécutables.

Nous présentons dans le Tableau I-2 des représentants de LDP formels.

Approche	Année	LDP	AGL-P	Référence
Procédurale	1995	APPL/A	Arcadia	[Sutton95]
	1994	PWI PML	PADM	[Bruynooghe94]
	1997	JIL	Julia	[Sutton97]
	1997	PBOOL	Rhodes	[Crégut97]
Déclarative	1988	MSL	Marvel	[Kaiser88]
	1988	Grapple	Grapple	[Huff88]
	1994	Merlin/PML	Merlin	[Junkermann94]
	1994	Spell	Epos	[Conradi94a]
	1994	Adele -Tempo	Adele -Tempo	[Belkhatir94]
Fonctionnelle	1989	HFSP	-	[Katayama89]
Basée Graphe/Réseau	1990	Melmac	Melmac	[Deiters90]
	1994	Slang	Spade	[Bandinelli94]
	1996	PROGRESS	Dynamite	[Heiman96]
	1998	APEL	-	[Dami98]
Basée UML	1998	Dynamite	-	[Schleicher98]
	1999	PROMENADE	-	[Franch99]
	2002	UML-based Process Language	-	[Chou02]
	2002	Executable PML from UML	-	[DiNitto02]
	2005	UML4SPM	UML4SPM	[Bendraou05]

Tableau I-2. Exemples de LDP formels

À cause de son caractère impératif, l'approche procédurale est trop rigide pour permettre l'évolution dynamique et incrémentale de modèles de procédé. A l'inverse, l'approche déclarative et l'approche fonctionnelle tolèrent l'indéterminisme des procédés, mais elles supportent mal la structuration de procédés. Quant à l'approche basée sur les graphes ou réseaux transitionnels, elle favorise la description de l'ordonnancement des activités et facilite l'évolution dynamique de procédés, mais elle ne représente pas bien l'aspect organisationnel des procédés. L'approche basée sur UML profite des notations standardisées, mais elle a besoin d'être renforcée par une sémantique opérationnelle pour devenir formelle.

I.4. MÉTHODES DE MODÉLISATION DE PROCÉDÉS

La section précédente nous a permis de synthétiser des travaux sur la représentation de modèles de procédés. Dans cette section nous nous intéressons à la conception de modèles de procédé. Nous étudions donc des travaux qui proposent des méthodes pour élaborer des modèles de procédé.

Du point de vue de l'ingénierie, l'élaboration et l'assistance à l'évolution de modèles[1] de procédé logiciel[2] est aussi un procédé. Ce procédé est appelé *méta-procédé* [Finkelstein94][Derniame99].

Dans la suite, nous discutons la définition de la notion de méta-procédé et présentons les travaux existants relatifs aux méta-procédés.

I.4.1. Notion de Méta-procédé

Bien que les définitions de méta-procédé dans [Finkelstein94] et [Derniame99] soient légèrement différentes, fondamentalement, un méta-procédé est un procédé pour élaborer, raffiner, modifier, et contrôler les modèles de procédé.

Activités du méta-procédé

Un méta-procédé est composé de plusieurs méta-activités qui manipulent les modèles de procédé. D'après [Conradi94b], il n'est pas possible d'identifier un méta-procédé universel pour tous les procédés logiciels, car il y a plusieurs façons de construire et de composer les méta-activités. Cependant, il est possible de définir certaines méta-activités de base qui constituent le squelette de tout méta-procédé.

Les méta-activités généralement admises [Finkelstein94][Derniame99] sont les suivantes :

- **Analyse des besoins de procédé** (Process Requirements Analysis) : Cette méta-activité a pour objet de fixer l'objectif de la modélisation et d'analyser les caractéristiques d'un procédé à modéliser.
- **Conception du procédé** (Process Design) : cette méta-activité est effectuée pour créer statiquement le modèle d'un nouveau procédé ou pour modifier le modèle d'un procédé existant. Le modèle résultant de cette activité peut comporter plusieurs sous-modèles selon la méthode de conception.
- **Implémentation du procédé** (Process Implementation) : Cette méta-activité a pour but d'instancier un modèle de procédé et d'exécuter cette instance, autrement dit d'assister le processus correspondant.
- **Évaluation du procédé** (Process Assessment) : cette méta-activité fournit des informations quantitatives et qualitatives concernant la performance d'un procédé. Ces informations peuvent permettre de définir de nouveaux besoins pour améliorer le procédé.

---

[1] L'évolution de modèles peut être le changement de contenu des modèles et/ou le changement de niveau d'abstraction de modèles (de conceptuel à exécutable, par exemple).

[2] Le terme « procédé de production » est parfois utilisé comme synonyme de procédé logiciel, c'est le procédé qui correspond aux activités de production et de maintenance de logiciels [Conradi91].

Besoins attendus d'un méta-procédé

En général, une méthode d'élaboration et d'évolution de procédé devra satisfaire les besoins suivants [Avrilionis95][Derniame99]:

- *Couvrir toutes les étapes* de l'élaboration du modèle de procédé ;
- *Supporter l'évolution* statique et dynamique de modèles ;
- *Prendre en compte la réutilisation* de modèles de procédés ;
- *Faciliter l'apprentissage et l'utilisation* du méta-procédé par une description simple et claire ;
- *Faciliter la réalisation* du méta-procédé par une description réalisable et complète.

Pour être complet, un méta-procédé doit guider l'élaboration (statiquement) et l'évolution (dynamiquement) de modèles tout au long de leur vie. Cependant, dans le cadre de cette thèse, nous mettons l'accent sur le support statique du méta-procédé. Le terme «méta-procédé» est donc utilisé dans le sens d'un procédé qui guide la modélisation de procédés.

I.4.2. Méta-procédés existants

Nous précisons d'abord la sémantique de la modélisation de procédés. Si le procédé à modéliser existe, la modélisation comprend la représentation de ce procédé dans un LDP approprié. Si la connaissance de développement n'est pas structurée en tant que procédé, la modélisation comprend la définition d'un procédé et la représentation du procédé défini sous forme de modèle(s).

Dans les deux cas, modéliser un procédé comporte des activités d'élicitation et de transformation de connaissances tacites de procédé en connaissances explicites. Ces activités sont complexes, la nécessité des méthodes pour guider la modélisation de procédés est donc évidente [Madhavji90][Nguyen94]. Cependant, à ce jour, peu d'efforts ont été faits sur ce sujet.

Les guidages méthodologiques concernant les procédés peuvent être classés en deux catégories : les méthodes de modélisation de procédés, et les standards d'évaluation et d'amélioration de procédés [Derniame99].

Méthodes de modélisation de procédés

Ce sont des méta-procédés qui définissent comment élaborer et faire évoluer les modèles de procédé. Ces méthodes proposent un ensemble de méta-activités et une démarche pour guider la modélisation et éventuellement l'exécution de procédés.

Plusieurs ateliers de génie logiciel centrés procédé disposent de leur propre méthode de modélisation et d'exécution de modèles de procédé (par exemple, SPADE[Bandinelli94], EPOS [Conradi94a], PWI [Bruynooghe94]), mais ces méta-procédés sont généralement implicites et codés en dur.

Parmi les travaux les plus significatifs au niveau méthodologique, on peut citer les projets PRISM[Madhavji90], ProcessEngineeringFramework [Kawalek94], Elicit[Turgeon96], la proposition de [Hollenbach96], les approches PERFECT[Birk97] et RHODES[TranDT01].

Les méthodes proposées dans les travaux ci-dessus sont représentées sous forme de méta-procédés qui couvrent les principales étapes de développement de modèles de procédé. PRISM et RHODES favorisent les modèles prescriptifs, tandis que ProcessEngineeringFramework, Elicit et PERFECT se focalisent sur les modèles descriptifs.

Méthodes d'évaluation de la qualité des procédés

Les travaux classés dans cette catégorise fournissent des guidages méthodologiques pour améliorer les procédés par l'évaluation de leur niveau de maturité[1]. Bien que leur objectif primaire ne soit pas le développement de modèles de procédé, ils préconisent certaines exigences ou activités de l'ingénierie des procédés concernant la définition et la modélisation de procédés.

Le travaux les plus représentatifs de cette catégorise sont la norme IEEE 1047 [IEEE97], les procédés organisationnels de la norme ISO/CEI 15504 (ORG 1-5 [ISO98]), les secteurs clés OPF (Organisational Process Focus), OPD (Organisational Process Definition) concernant la gestion de procédés organisationnels du modèle CMMI [CMMI02].

I.5. SYNTHÈSE

Nous avons décrit brièvement dans cette section les différentes approches de description de procédés ainsi que les principales méthodes de modélisation de procédés.

La première observation que l'on peut tirer de cette vue d'ensemble est qu'il existe un grand nombre de LDP qui ne diffèrent que légèrement, que ce soit aux niveaux conceptuel ou notationnel. Nous remarquons également que les LDP de la deuxième génération ont tendance à utiliser la méta-modélisation[2] pour définir leur modèle conceptuel, et la notation UML pour représenter les modèles de procédé. La conclusion qui ressort de ces observations est qu'une standardisation des LDP pour faciliter la communication de procédés est nécessaire. L'apparition de la norme SPEM [SPEM05] de OMG confirme cette nécessité.

Quant à la méthodologie de modélisation, il nous semble que les méthodes existantes sont insuffisamment développées et décrites pour guider pratiquement les concepteurs de procédé dans leur travail. A notre avis, les méthodes de modélisation devraient être plus détaillées et décrites plus formellement pour qu'elles puissent être exploitées dans les AGL-Ps pour aider les concepteurs de procédés.

II. RÉUTILISATION DE PROCÉDÉS

Les procédés logiciels sont centrés humain, enclins à changer, très flexibles, et plutôt instables [Derniame99]. Cette nature complexe rend les procédés logiciels difficiles à décrire et à contrôler. En conséquence, la modélisation de procédés[3] est une tâche qui demande des efforts et des investissements importants, et qui se traduit souvent par des dépassements de temps et de budget [Franch99][Godart99]. Augmenter la productivité et accroître la qualité des procédés logiciels sont donc des objectifs importants de l'ingénierie des procédés [Derniame99][Derniame04]. L'approche

[1] La maturité du procédé logiciel caractérise dans quelle mesure un procédé est explicitement défini, géré, mesuré, et efficace.

[2] La méta-modélisation est une technique permettant de définir les langages de modélisation avec un modèle explicite.

[3] Pour simplifier, les procédés considérés dans la suite sont implicitement des procédés logiciels

de construction de procédés par réutilisation est parmi celles contribuant à ces objectifs [ISPW96]. Elle se définit comme une approche où la construction d'un procédé s'appuie sur des connaissances de procédé existantes, éprouvées et réutilisables [Hollenbach96].

Il est usuel, dans le domaine de la réutilisation, de distinguer deux types de problématiques de recherche [Fugini92][Mili95] : celle qui relève de *l'ingénierie pour la réutilisation* et celle concernant *l'ingénierie par la réutilisation* ; la première consiste à identifier, représenter et organiser la connaissance alors que la seconde concerne la construction d'un système par réutilisation de cette connaissance. La Figure I-2 montre l'adaptation de ces deux types d'ingénierie au contexte des procédés.

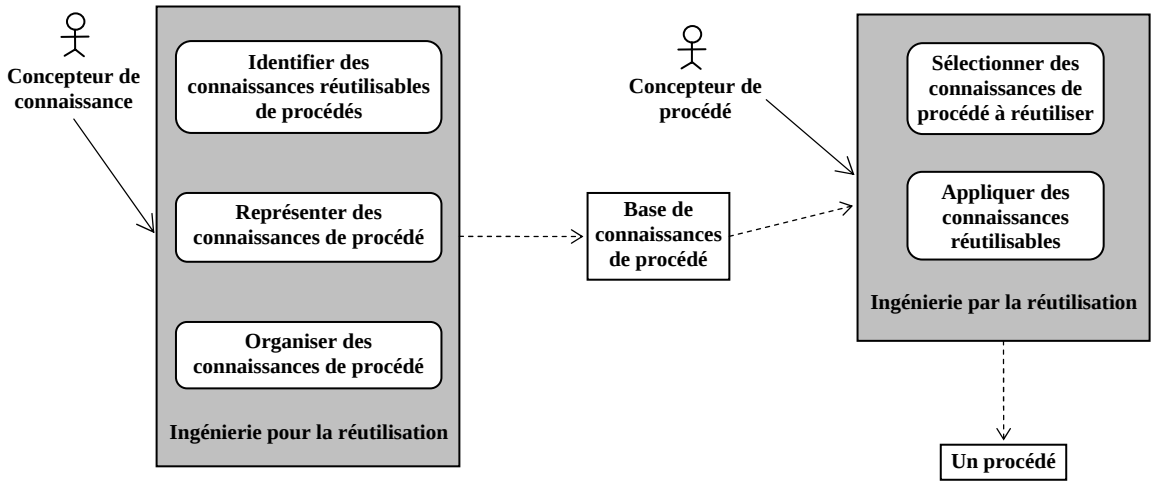


Figure I-2. Ingénierie pour et par la réutilisation dans le contexte des procédés

Nous discutons dans la suite des différentes facettes de ces deux types d'ingénierie dans le contexte de la réutilisation de procédés. Pour cela, nous proposons un cadre de référence basé sur la classification par facettes proposée dans [Priéto87] pour évaluer les approches de réutilisation de procédés. Le cadre de référence propose, pour chaque facette, un ensemble de critères (attributs) dont les valeurs permettent d'évaluer les travaux sur la réutilisation de procédés.

II.1. PROBLÉMATIQUE DE L'INGÉNIERIE POUR LA RÉUTILISATION DE PROCÉDÉS

L'objectif de *l'ingénierie pour la réutilisation de procédés* est de développer des méthodes supportant la production des connaissances de procédé réutilisables. La problématique de l'ingénierie pour la réutilisation de procédés concerne trois facettes :

(i) l'identification des connaissances candidates à la réutilisation;

(ii) leur représentation sous forme d'artéfacts réutilisables ;

(iii) leur organisation au sein de systèmes de réutilisation.

Dans la suite, nous analysons ces facettes et identifions leurs valeurs possibles.

II.1.1. Identification des connaissances de procédé

Différents types d'informations réutilisables de procédés sont identifiés et discutés dans plusieurs travaux [Basili91][Arbaoui96][Kellner96][Hollenbach96][Groenewegen96] [Jørgensen00][Neu03]. Ces informations sont considérées comme des *connaissances de procédé*[1].

Nous proposons deux attributs pour caractériser les connaissances de procédé : la *nature* de la connaissance et la *couverture* de la connaissance.

Nature d'une connaissance de procédé

Les connaissances de procédé sont des savoirs et des savoir-faire acquis pendant la définition de procédés. Elle englobe des résultats de définition de procédé, et des expériences sur la définition de procédé. Nous distinguons donc deux formes de nature de connaissance de procédé :

- **Connaissances de procédé de développement** : Ces sont des connaissances relatives à la conception et à la production de produits logiciels. Ce type de connaissance est inclus dans les résultats de la définition de procédés c'est-à-dire les procédés logiciels eux-mêmes. De telles connaissances concernent la spécification des éléments de procédé (c.f. I.2), la définition du contenu et de la réalisation de ces éléments (par exemple la décomposition d'un produit, les actions d'une tâche), et des relations entre eux (par exemple l'ordre d'exécution des tâches, les relations d'impact entre produits).

  Parmi les travaux les plus connus qui décrivent la connaissance de procédé de développement, on peut citer les modèles de développement comme Spiral [Boehm88], VMethode [SFB501], et Unified Process [Jacobson99], les normes ISO12207[ISO95], PSS-05[Mazz94], le modèle de maturité CMMI [CMMI02].

- **Connaissances de modélisation de procédés :** Ces sont des connaissances relatives à la définition de procédés[Rupprecht00]. Plus concrètement, ce sont des expériences accumulées durant la modélisation de procédés. Elles reflètent les solutions techniques ou méthodologiques pour décrire les procédés. Ces connaissances seront très utiles si elles peuvent être identifiées, formalisées, structurées et mises à la disposition des concepteurs de procédé.

  Dans ce type de connaissance, on peut trouver par exemple des méta-procédés (comme PRISM[Madhavji90], PERFECT[Birk97]), des solutions de description de procédés (par exemple les Workflow Patterns [v.d.Aalst03]) ou des solutions d'organisation et de structuration de procédé (par exemple [Harrison96][Malone03][Penker00]).

Couverture d'une connaissance de procédé

La couverture d'une connaissance correspond aux domaines dans lesquels sa réutilisation est possible, voire pertinente. Nous distinguons trois types de couverture de connaissance pour les procédés :

- **Dépendante d'un projet :** Les connaissances ayant cette couverture ne sont réutilisables, telles quelles, que pour un nombre restreint de projets ayant un ensemble commun de caractéristiques, notamment les produits.

[1] En anglais on parle de «process asset» ou «process experience».

Par exemple, la connaissance de procédé pour produire les logiciels impliqués dans l'équipement aéronautique d'Airbus est dépendante d'un projet.

- **Dépendante d'un domaine :** Les connaissances ayant cette couverture sont réutilisables au sein des procédés d'un même domaine, quel que soit leurs produits spécifiques.

  Par exemple, la procédure de révision d'artéfacts logiciels peut s'appliquer dans tous les procédés logiciels sur des produits différents ; les démarches de vente d'un produit peuvent être appliquées pour vendre un produit quelconque.

- **Indépendante du domaine :** Les connaissances ayant cette couverture peuvent être réutilisées au sein de n'importe quel procédé, quel que soit son domaine. Ce sont des connaissances de procédé génériques.

  Par exemple, une solution pour synchroniser des activités parallèles peut être employée aussi bien pour organiser un procédé logiciel qu'un procédé métier.

II.1.2. Représentation des connaissances de procédé réutilisables

Une fois les connaissances utiles identifiées, il faut trouver un moyen pour les représenter sous une forme réutilisable.

La modélisation de procédés a pour rôle principal d'expliciter les connaissances en développement (c.f. I). Quant aux connaissances relatives à la définition de procédés, elles sont généralement représentées de façon informelle sous forme de conseils ou de guidages (comme dans [Coplien94][Foote95][Malone03]).

Sur la base des besoins attendus d'un formalisme de description de procédés (c.f. I.3.1) et de la caractérisation présentée ci-avant (c.f. II.1.1), nous identifions ci-dessous les attributs importants d'une représentation favorisant la réutilisation de procédés :

Expressivité

Cet attribut caractérise la capacité de représentation de la connaissance de procédé d'un formalisme. Nous distinguons deux niveaux d'expressivité selon que l'on considère :

- **Les éléments de procédé :** un formalisme à ce niveau permet de représenter les éléments de procédé et leurs relations.
- **Les éléments de procédé réutilisables :** un formalisme à ce niveau permet de représenter non seulement les éléments de procédé, mais aussi les éléments réutilisables ainsi que la manière de les réutiliser (appliquer) dans la description de procédés.

Modularité

La représentation modulaire de connaissances rend leur application flexible et sélective. Elle permet la définition de procédés par l'assemblage de fragments de procédé existants. En considérant cet attribut, un formalisme peut avoir l'une des valeurs suivantes :

- **Absence de mécanisme de modularisation** : le formalisme ne supporte pas la modularité.
- **Existence d'un mécanisme de modularisation** : le formalisme supporte la modularité.

Abstraction

Cet attribut a pour but d'indiquer si un formalisme permet de représenter les connaissances de diverses couvertures pour promouvoir la réutilisation de procédés. Les valeurs possibles de cet attribut sont :

- **Un niveau d'abstraction** : le formalisme supporte la représentation de procédés à un seul niveau d'abstraction.
- **Plusieurs niveaux d'abstraction** : le formalisme supporte la représentation de procédés à différents niveaux d'abstraction.

Standardisation de la représentation

La compréhensibilité d'un formalisme dépend du type d'interface et du degré de standardisation de concepts et de notations utilisées (c.f. I.3.1). Dans le cas de connaissances réutilisables, l'aspect standardisation de la représentation est important pour faciliter la communication des connaissances représentées. Les valeurs associées à cet attribut sont ainsi :

- **Non-standardisée** : les concepts et les notations du formalisme sont spécifiques.
- **Standardisée** : le formalisme utilise des concepts et des notations standardisés.

Formalisation

Le degré de formalisation d'un formalisme est important dans le cas des connaissances réutilisables car il conditionne le développement d'outils de réutilisation de procédés. Nous retenons les valeurs classiquement adoptées pour cette formalisation :

- **Informelle :** le formalisme est intuitivement défini.
- **Semi-formelle :** le formalisme dispose d'une syntaxe formelle.
- **Formelle :** le formalisme dispose d'une syntaxe formelle et d'une sémantique formelle.

II.1.3. Organisation des connaissances de procédé réutilisables

Pour faciliter la sélection, l'adaptation et l'application de connaissances réutilisables, une approche de réutilisation devra fournir une solution pour d'un côté, modulariser les connaissances, de l'autre établir les liens entre les modules de connaissance.

Nous proposons donc les deux attributs suivants pour caractériser cette facette :

Type d'encapsulation de modules de connaissance

Cet attribut caractérise le type d'encapsulation des connaissances capturées. Nous distinguons deux types de conteneur de connaissances de procédé réutilisables :

- **Composant de procédé** : un composant de procédé encapsule une connaissance de développement en utilisant le principe de la boîte-noire (black-box). Il fournit donc un fragment de procédé autonome sans montrer son contenu. Aucune connaissance de guidage n'est généralement associée au composant pour expliquer son utilisation. On peut citer certains travaux sur les composants de procédé comme [Avrilionis96][Lindquist97] [Bergner98][TranDT01].

- **Patron de procédé :** un patron de procédé capitalise la connaissance de procédé en utilisant le principe de la boîte-blanche (white-box) pour apporter une guidage méthodologique à la définition de procédé. Il décrit explicitement la connaissance capturée ainsi que la manière de l'appliquer. Les patrons de procédé sont étudiés dans des travaux comme [Buschmann96][Ambler98][Vasconcelos98][Firesmith01].

Structuration des connaissances

Cet attribut caractérise l'organisation d'un ensemble de connaissances réutilisables pour faciliter la sélection des connaissances à réutiliser. Nous distinguons deux approches pour organiser les connaissances :

- **Bibliothèque de connaissances** : les entités réutilisables sont stockées dans une structure indicée qui permet de regrouper les connaissances selon leur sujet. C'est le cas des approches basées sur Experience Factory [Basili94] ou [Henninger98][CMMI02].
- **Carte de connaissances** : les relations entre différentes entités réutilisables sont explicitement définies. Grâce à ces relations, une carte des connaissances peut être établie pour faciliter la navigation dans les connaissances. Parmi les travaux adoptant cette approche, on peut citer [Ambler98][Conte02][Deneckere01][Firesmith01] .

II.2. PROBLÉMATIQUE DE L'INGÉNIERIE PAR LA RÉUTILISATION DE PROCÉDÉS

Supposons maintenant que les connaissances réutilisables aient été capturées sous forme d'entités réutilisables. L'objectif de *l'ingénierie par la réutilisation de procédés* est de développer des méthodes pour réutiliser ces entités dans le but de construire de nouveaux procédés ou d'enrichir des procédés existants. Nous distinguons trois facettes de la problématique de l'ingénierie par la réutilisation de procédés :

(i) opérateurs de manipulation d'entités réutilisables;

(ii) guidage méthodologique pour réutiliser systématiquement les entités réutilisables dans l'élaboration des modèles de procédé;

(iii) outils support pour faciliter l'application des entités réutilisables de procédé.

Dans la suite, nous proposons des attributs pour chaque facette et identifions leurs valeurs possibles.

II.2.1. Opérateurs de manipulation d'entités réutilisables

Pour faciliter l'élaboration de modèles basés sur des entités réutilisables, une approche par la réutilisation devra fournir des opérateurs permettant la manipulation (plus précisément la recherche, la sélection, l'adaptation et la composition) des entités réutilisables ainsi que du modèle construit. Nous proposons deux attributs pour caractériser cette facette :

Mécanisme de réutilisation

La réutilisation des connaissances contenues dans les entités réutilisables peut être réalisée par plusieurs mécanismes. Nous identifions trois façons de réutiliser des connaissances de procédé :

- **Spécialisation** : les connaissances fournies sont adaptées par spécialisation pour le procédé en cours de modélisation.
- **Paramétrisation** : l'entité à réutiliser est associée à un ensemble de paramètres. Lors de la modélisation d'un procédé, la réutilisation des connaissances capturée dans l'entité est réalisée par le choix de valeurs pour les paramètres. On peut dire que c'est une adaptation guidée et contrainte.
- **Composition** : les connaissances fournies sont associées les unes aux autres de façon cohérente afin de former un (fragment de) procédé complet.

Mode de définition

Cet attribut indique comment les opérateurs de manipulation sont intégrés à la représentation et à la manipulation des entités réutilisables. Nous distinguons deux modes de définition d'opérateurs :

- **Définition implicite** : les opérateurs sont définis implicitement et séparément du formalisme de description de procédés. Par conséquent, l'application d'entités réutilisables dans les modèles de procédé est implicite.
- **Définition explicite** : les opérateurs sont définis comme des éléments du langage de description de procédés. Par conséquent, on peut les utiliser pour élaborer les modèles de procédé basés sur les entités réutilisables. Autrement dit, leur application est explicite.

II.2.2. Guidage méthodologique

Pour guider les concepteurs dans la réutilisation systématique de connaissances existantes au cours de la modélisation de leurs procédés, un guidage méthodologique est nécessaire. Nous caractérisons cette facette par les attributs suivants :

Type de guidage

Cet attribut reflète la nature du guidage proposé. Nous distinguons deux types de guidage :

- **Conseils pour la réutilisation d'une entité :** le guidage décrit le contexte d'application d'une entité réutilisable et les situations recommandées pour la réutiliser.
- **Démarche de réutilisation** : le guidage propose une véritable démarche pour mettre en oeuvre la réutilisation.

Spécification du guidage

Cet attribut reflète la façon de décrire le guidage proposé. Nous distinguons deux niveaux de spécification de guidage :

- **Gros grains** : le guidage est décrit de manière générale, sans structuration particulière.
- **Grains fins :** le guidage est décrit de manière détaillée, éventuellement implantable.

II.2.3. Outils support

Des outils support peuvent supporter la modélisation de procédés par la réutilisation et ainsi augmenter la cohérence de l'application des entités réutilisables. Nous caractérisons cette facette par deux attributs :

Catégorie

Tous les outils supportant la modélisation offrent un certain nombre de fonctionnalités. Cependant, tous ne prennent pas en charge de façon équivalente la réutilisation d'entités. Deux catégories d'outils peuvent être identifiées avec cet objectif :

- **Gestion de bibliothèque de composants :** ce type d'outil fournit une collection de composants réutilisables. Il gère la persistance des composants et offre des interfaces de recherche qui exploitent, quand elle existe, l'organisation interne des composants. Dans cette approche, la sélection de composants, leur adaptation et leur intégration dans un procédé en cours de modélisation sont à la charge du concepteur.
- **Environnement de modélisation par la réutilisation** : ce type d'outil de modélisation de procédés intègre des fonctions de réutilisation. Il ne fournit pas seulement la gestion d'une collection de composants réutilisables, mais aussi des fonctions pour sélectionner, adapter et appliquer des composants réutilisables à un modèle de procédé.

Mode de support

Cet attribut reflète le niveau d'automatisation des outils. Nous distinguons les trois degrés d'automatisation suivants :

- **Manuel** : l'outil permet d'accéder aux entités réutilisables, mais leur réutilisation est entièrement à la charge de l'utilisateur.
- **Semi-automatique** : l'outil fournit des fonctions assistant l'application d'entités réutilisables mais nécessite cependant l'intervention de l'utilisateur.
- **Automatique** : l'application d'entités réutilisables est entièrement automatisable et peut donc se faire sans intervention de l'utilisateur.

II.3. SYNTHÈSE

La Tableau I-3 ci-dessous récapitule les critères relatifs à la réutilisation de procédés définis dans cette section ainsi que les facettes auxquelles ils appartiennent. Il constitue ainsi un cadre de référence qui servira à évaluer et comparer les approches de réutilisation à base des patrons qui seront présentées dans les sections III.3 et III.4.

Besoin	Facette	Attribut	Valeurs
Formalisation de connaissances réutilisables	Identification des connaissances	Nature	{ connaissance de développement, connaissance de modélisation de procédés}
		Couverture	{ dépendante d'un projet, dépendante d'un domaine, indépendante}
	Représentation des connaissance	Expressivité	{ éléments de procédé, éléments réutilisables}
		Modularité	{ non supportée, supportée
		Abstraction	{ un niveau d'abstraction, plusieurs niveaux d'abstraction}
		Standardisation	{ non-standardisée , standardisée }
		Formalisation	{ informelle, semi-formelle, formelle}
	Organisation des connaissance	Encapsulation	{ composant de procédé, patron de procédé }
		Structuration	{ bibliothèque de connaissance, carte de connaissance}
Application de connaissances réutilisables	Opérateurs de manipulation	Définition	{ implicite, explicite}
		Mécanisme	{ spécialisation, paramétrisation, composition}
	Guidage méthodologique	Type de guidage	{ conseils de réutilisation, démarche de réutilisation}
		Spécification	{ gros grains, grains fins}
	Outils support	Catégorie	{ gestion de bibliothèque de composants, environnement de modélisation}
		Mode de support	{ manuel, semi-automatique, automatique}

Tableau I-3. Récapitulatif des facettes et attributs du cadre de référence

III. PATRONS DE PROCÉDÉ

Inspirés par le travail d'Alexander sur les patrons d'architecture [Alexander79], K. Beck et W. Cuningham [Beck87] ont introduit en 1987 le concept de patron dans l'ingénierie du logiciel. Un patron logiciel décrit un *problème* fréquemment rencontré dans un *contexte* particulier de

développement ainsi qu'une *solution* générale et éprouvée pour résoudre ce problème [Coplien96]. De nos jours, la notion de patron est reconnue dans la communauté du Génie Logiciel comme un moyen efficace pour réutiliser des connaissances acquises lors de différentes phases du cycle de vie du développement. C'est un niveau de réutilisation qui vise à fournir une assistance méthodologique au développement.

Les patrons peuvent être classés selon le type de connaissance représentée en deux catégories [Conte01] : les *patrons de produit* capitalisant des spécifications ou des implémentations des résultats de développement, les *patrons de procédé* capitalisant des spécifications ou des implémentations de démarches à suivre pour réaliser le développement.

Les patrons de produit sont aujourd'hui connus et largement utilisés. Parmi les travaux les plus significatifs, nous citons les patrons d'analyse de Coad [Coad95] et Fowler [Fowler97]; les patrons de conception de Gamma [Gamma94] ; les patrons d'architecture de [Buschmann96] et les patrons d'implémentation de Coplien [Coplien92].

Les patrons de procédé, quant à eux, constituent encore un sujet émergent et non complètement maîtrisé. Dans la suite, nous introduisons d'abord la notion de patron de procédé en examinant les définitions existantes (III.1) et un ensemble de patrons réels (III.2). Nous nous intéressons ensuite à la formalisation (III.3) et l'utilisation (III.4) des patrons de procédé dans les travaux relatifs.

III.1. NOTION DE PATRON DE PROCÉDÉ

III.1.1. Définitions existantes

Le terme "*patron de procédé*" (process pattern) a été introduit la première fois à la conférence PLoP'94 par Coplien [Coplien94] dans le sens d'un patron qui capture des connaissances de procédé[1]. Depuis, plusieurs travaux dans les domaines des procédés logiciels, des procédés métier, de l'ingénierie de méthodes et des workflows ont employé cette notion en tant que moyen de capitalisation et de réutilisation des procédés.

Nous citons ici quelques définitions de patron de procédé trouvées dans la littérature de l'ingénierie des procédés. En général, un patron de procédé est défini comme un patron capturant *un procédé éprouvé* [Dittmann02][Hagen04], ou *un ensemble de techniques et d'activités* [Ambler98][Bergner98] applicables pour résoudre *un problème récurrent de développement* . D'autres auteurs définissent un patron de procédé comme un modèle qui fournit *une conception généralisée pour les procédés* [Iida02], ou un modèle qui capture *une structure commune de procédé* [Storrle01].

III.1.2. Intérêt des patrons de procédé

Comme tous les types de patrons, les patrons de procédé permettent la capture de connaissances des procédés éprouvés, l'abstraction de problèmes et de solutions de développement, et facilitent la communication ainsi que la compréhension de procédés [Dittmann02].

[1] Cependant, les patrons de Coplien sont plutôt des patrons d'organisation décrivant les structures techniques et humaines d'une organisation pour gérer le développement.

De plus, les patrons de procédé peuvent servir comme moyen efficace pour modéliser les procédés [Bergner98][Storrle01] car ils permettent de :

- *décrire les procédés de manière modulaire et réutilisable* : un patron capture un fragment de procédé réutilisable correspondant à un problème spécifique. Un procédé peut être décrit en composant plusieurs patrons.
- *décrire les procédés de manière flexible et adaptable :* un patron est applicable dans plusieurs situations, il peut donc contribuer à rendre les modèles de procédé plus flexibles et plus adaptables.

III.1.3. Bilan

Les patrons de procédé semblent prometteurs pour la réutilisation de procédés. Pourtant, les définitions existantes sont encore imprécises et ne font pas l'objet d'un consensus. Le contenu exact d'un patron de procédé reste encore vague. Nous pouvons tirer toutefois l'idée commune suivante des définitions existantes : *un patron de procédé décrit un ensemble d'activités qui peuvent être exécutées dans une situation spécifique pour résoudre un problème récurrent de développement*.

III.2. PATRONS DE PROCÉDÉ EXISTANTS

Pour mieux comprendre la notion de patron de procédé, nous avons essayé d'observer des patrons de procédé réels. Plusieurs travaux ont tenté d'extraire des connaissances de procédé à partir de procédés éprouvés ; cependant, en cherchant des systèmes explicitement dédiés aux patrons de procédé, nous n'avons trouvé que le travail d'Ambler [Ambler98][Ambler99]. Ce résultat apparemment étonnant provient d'une certaine confusion dans la terminologie utilisée par les différents auteurs[1].

Aussi, pour avoir une vue plus complète des approches relatives aux patrons de procédé, nous élargissons notre étude aux travaux existants[2] significatifs dans le domaine des connaissances de procédé réutilisables.

III.2.1. Le langage de patrons de procédé OOSP

Dans [Ambler98][Ambler99], Ambler a introduit le procédé de développement orienté-objet OOSP (Object Oriented Software Process). Ce procédé est représenté comme une collection de patrons de procédé qui adressent les problèmes spécifiques du développement logiciel selon l'approche orientée-objet (Figure I-3).

Ambler distingue trois types de patrons de procédé :

- **Patron de tâche** (Task Pattern) : Ce type de patron de procédé décrit les activités pour accomplir une tâche spécifique ; on peut citer par exemple le patron *RequirementTesting*.

[1] Il existe plusieurs termes relatifs qui recouvrent des significations assez voisines (par exemple, process pattern, business process pattern, workflow pattern).

[2] Les systèmes sont sélectionnés non seulement dans le domaine des procédés logiciels, mais aussi dans le domaine des procédés métiers et des systèmes d'information.

- **Patron d'étape** (Stage Pattern) : Ce type de patron de procédé décrit les tâches, souvent exécutés itérativement, d'une étape de projet. Un patron d'étape peut être composé de plusieurs patrons de tâche. Par exemple, le patron d'étape *Program* comprend les patrons de tâches *Write Source Code, Reuse existing components, Technical Review, etc.*
- **Patron de phase** (Phase Pattern) : Ce type de patron de procédé décrit les interactions entre les patrons d'étape pendant une phase d'un projet. Un patron de phase est un ensemble de patrons d'étape. Par exemple, le patron de phase *Construct* comprend les patrons d'étape *Model, Program, Test In The Small* et *Generalize*.

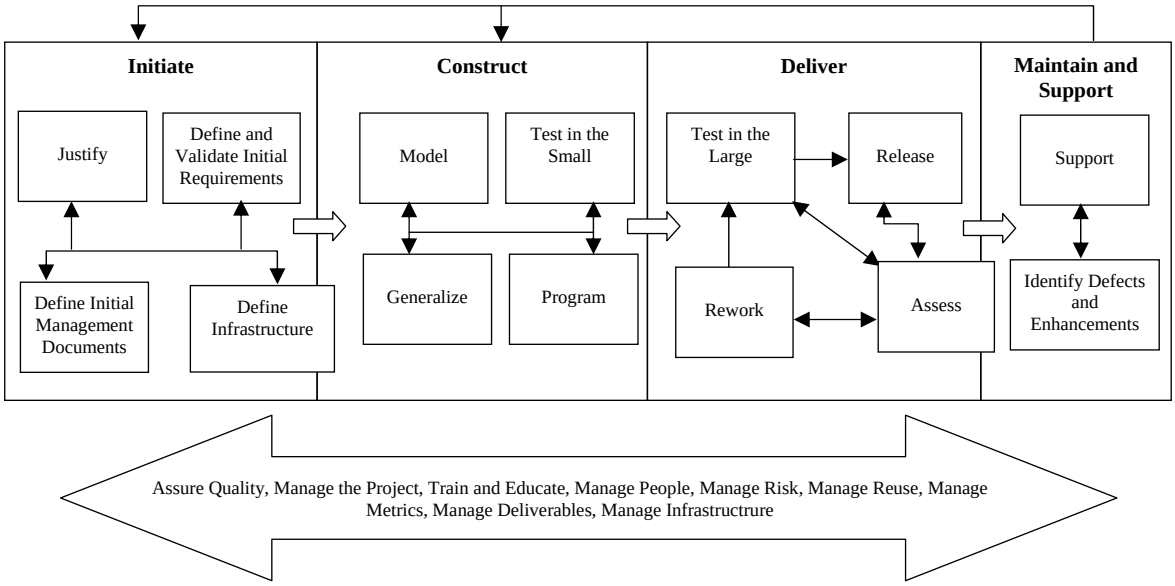


Figure I-3. Procédé OOSP

Pour chaque patron d'étape, Ambler indique les patrons de tâche à appliquer dans l'étape. En outre, plusieurs patrons de tâches qui doivent être appliqués tout au long du développement sont également décrits. Par contre, alors que les patrons de phase et les patrons d'étape sont définis et représentés dans un formalisme structuré, les patrons de tâches ne sont pas identifiés et décrits avec précision.

III.2.2. Open Process Framework (OPF)

OPF [Firesmith01] est un cadre de processus qui offre un support méthodologique pour le développement orienté-objet. OPF propose une base de *composants de méthode* (Method Component) *réutilisables* qui peuvent être instanciés et adaptés pour construire des procédés orientés-objets. La Figure I-4 montre six types de composant de méthode définis dans OPF sous forme de classes et les relations entre eux :

- **Producer** : cette classe représente différents types d'agent qui réalisent le développement. Ses sous-classes principales sont : *Organization*, *Team, Role, Person, Tool.*
- **Work Unit** : cette classe représente des travaux effectués pour manipuler des artefacts de développement. Ses sous-classes sont *Activité, Task, Technique* et *Workflow*. Une activité

se compose d'un ensemble cohésif de tâches qui sont organisées selon un flux de contrôle. Une technique représente une façon de réaliser une tâche, applicable donc à plusieurs tâches.

- **Work Product** : cette classe représente les artéfacts utilisés ou créés durant le développement. Elle est spécialisée en plusieurs sous-classes telles que *Application*, *Component, Diagram, Metric, Pattern, Test.*
- **Stage** : cette classe représente les éléments d'organisation de développement. Un stage modèlise une synchronisation temporelle de l'exécution d'un ensemble de WorkUnits. La classe *Stage* peut être spécialisée en sous-classes telles que *Cycle*, *Phase*, *Build* et *Milestone*.
- **Endeavor** : cette classe représente des plans de développement. Elle est spécialisée en trois sous classes : *Entreprise*, *Program* et *Projet*.
- **Language** : cette classe représente les différents langages utilisés dans le développement. Ses sous-classes principales sont : *Constraint Language*, *Implementation Language, Modeling Language, Natural Langue, Specification Language*.

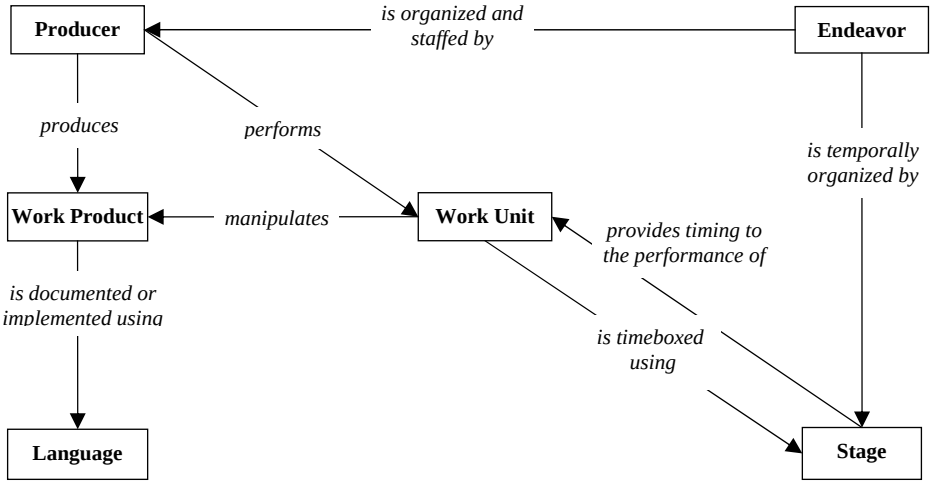


Figure I-4. Composants principaux d'OPF

III.2.3. Le langage de patrons de Bergner

Bergner et al. [Bergner98] ont proposé un langage de patrons de procédé permettant de développer des procédés logiciels orientés-composant. Ce langage propose quatre catégories de patrons :

- **Patrons de Projet** (Project Patterns) : ces sont les patrons qui fournissent des cycles de vie de procédés. Autrement dit, ils décrivent des ordres de réalisation des phases du procédé pour créer les artefacts principaux (MainResult). Citons dans ce groupe les patrons *Topdown, BottomUp, ArchitectureDriven, Roundtrip, Iterative.*
- **Patrons de Résultats Intermédiaires** (Inter-Result Patterns) : un patron de cette catégorie suggère une solution qui produit des artefacts dans au moins deux phases différentes de développement. Par exemple, les patrons *TechnicalFoundation, Combined Experimental Prototyping, Component Assessment et Component Update* sont des patrons Inter-Result.

- **Patrons de Résultats Primaires** (Main Result Patterns) : ces patrons concernent la manière d'élaborer les artefacts principaux du développement. Nous avons comme exemples de ce groupe les patrons *Customer-Driven Analysis , Market-Driven Analysis, Experimental Prototyping, Design-Driven Evaluation*.
- **Patrons de Sous-Résultats** (Subresult Patterns) : Ce type de patron fournit des solutions de création de sous-artefacts (subresult) dans une phase de développement. Parmi ces patrons on peut citer *Adaptation by Wrapping* et *Adaptation by ReImplementation*.

La Figure I-5 montre une partie du patron *Architecture-Driven* de Bergner.

Name	**Architecture-Driven (Project Pattern)**

Intent: Organizing the overall development process by first establishing a system architecture. Starting out from this architecture, the developer tries to gather reasonable user requirements, designs appropriate business as well as technical components and builds an implementation strongly based on this architecture.

Tasks and Roles:

- The *System Architect* has a clear understanding of the available technical and business-oriented architectures in the given application domain. He is responsible for searching and evaluating these architectures and the appropriate components.
- After a certain technical and business-oriented architecture has been chosen, the *System Architect* prepares the according design documents which serve as input to the results of Analysis, System Specification and finally Implementation.
- While elaborating the Analysis documents, a product profile needs to be established according to the market demands.
- Once tentative requirements based on the given system architecture have been identified, it is possible to elicit concrete user requirements and supply them to the other development tasks.
- Specification requires the integration of both technical and business-oriented architecture to produce a complete and consistent specification document which allows implementation of the system.
- The available components are mainly tested against predefined requirements of the chosen system architecture until the actual user requirements are provided by the results of Analysis.

Related Patterns Variants of this pattern are **Technical Architecture Driven** and **Business Architecture Driven** which focus on a dedicated part of the overall system architecture. The patterns **Experimental Business Prototyping**, **Experimental Technical Prototyping**, and **Combined Experimental Prototyping** may be applied to understand as well as to evaluate the conceivable architectures and available components.

Figure I-5. Extrait du patron Architecture Driven [Bergner98]

III.2.4. Le Catalogue SIP

Gzara a proposé un catalogue de patrons spécifiques pour les Systèmes d'Information de Produits (SIP) [Gzara00][Gzara03]. Les patrons de ce catalogue couvrent les étapes d'analyse et de conception des SIP en combinant *patrons de produit* et *patrons de procédé*. Les patrons SIP sont organisés selon les trois catégories suivantes :

- des **patrons d'analyse de produit** qui fournissent des fragments de modèle pour représenter les produits.
- des **patrons d'analyse de processus** qui offrent une manière pour représenter et décomposer les processus métier du SIP. Ils permettent d'analyser les processus en mettant en évidence les activités qui les composent, les éléments entrants et sortants mis en jeu ainsi que les ressources employées dans chacune des activités.

- des **patrons de conception** qui offrent une démarche permettant d'identifier, à partir de la décomposition des processus métiers, les objets du Système d'Information Informatique (SII) afin d'aboutir au modèle de conception du SIP.

Dans la Figure I-6 nous montrons un exemple de patron SIP.

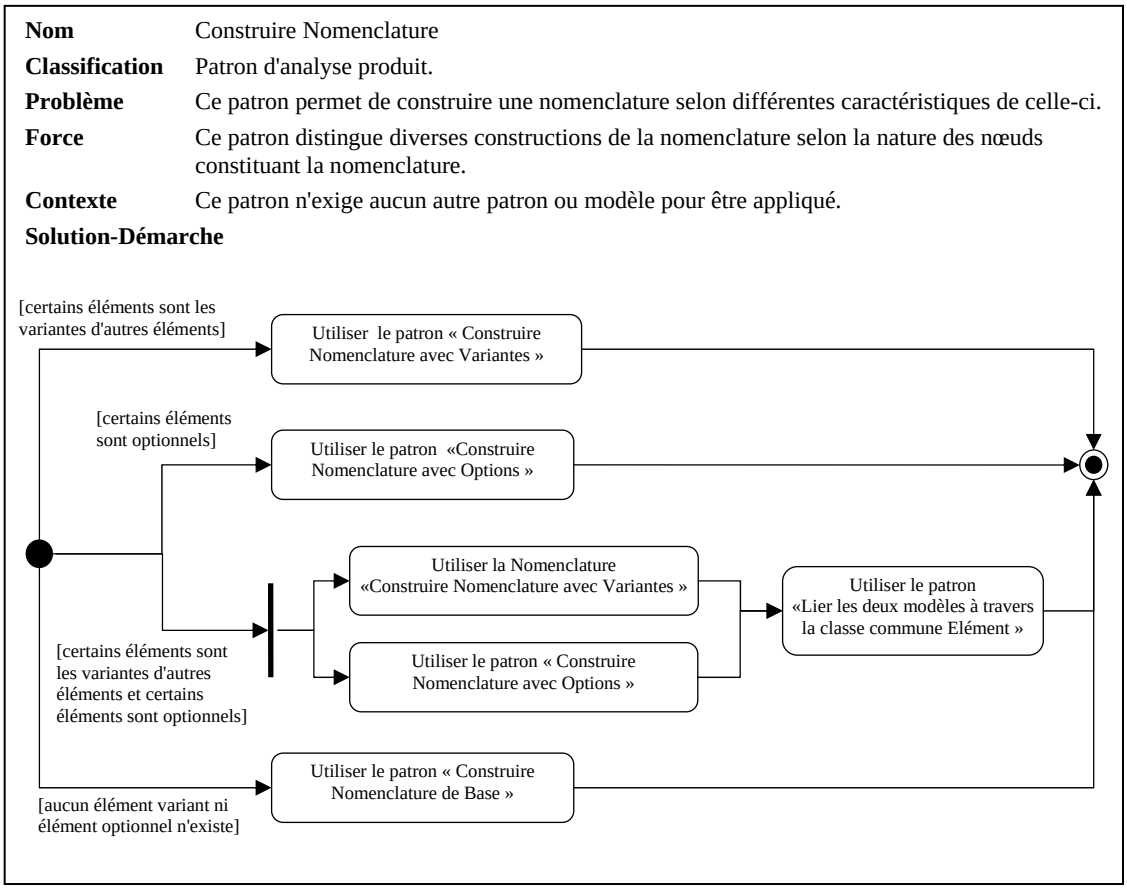


Figure I-6. Patron Construire Nomenclature de SIP [Gzara00]

III.2.5. WorkFlow Patterns de Van der Aalst

Dans la communauté des procédés métier (Business Processes), il existe le concept de patron de workflow (Workflow Pattern) proposé par Van der Aalst et al. [v.d.Aalst03]. Un patron de workflow propose une solution pour représenter un flux de contrôle[1] entre des activités dans un contexte donné. Bien que les patrons de workflow ne décrivent d'activité réutilisable comme les patrons de procédé logiciel vus précédemment, nous les considérons comme une sorte de patrons de procédé qui décrivent des connaissances de modélisation de procédés.

Les patrons de workflow sont divisés en six groupes :

- les patrons **Basic Control Flow** décrivant des flux de contrôle de base entre activités (par exemple : *Sequence, Parallele Split, Exclusive Choice, Simple Merge*);

[1] control flow

- les patrons **Advanced Branching and Synchronization** décrivant des enchaînements complexes et synchronisés (par exemple : *MultipleChoice, SynchronizingMerge)* ;
- les patrons **Structural** représentant des situations concernant les contraintes structurelles imposées aux procédés (par exemple : *Arbitrary Cycles, Implicit Termination*) ;
- les patrons **Multiple Instances** décrivant des solutions pour des situations où plusieurs instances d'une activité sont actives en même temps (par exemple : *Multiple Instances Without Synchronization, Multiple Instances With a priori Design Time Knowledge*) ;
- les patrons **State-based** représentant des enchaînements d'activités basés sur les états d'exécution des activités (par exemple : *Deferred Choice, Interleaved Parallel Routing*) ;
- les patrons **Cancellation** représentant des constructions pour modéliser l'annulation d'une activité ou d'un procédé, comme par exemple *Cancel Activity*.

Les patrons de workflow de Van der Aalst fournissent un ensemble de constructions génériques qui permettent de modéliser la plupart des enchaînements d'activités des procédés. Nous montrons quelques patrons de workflow dans la Figure I-7.

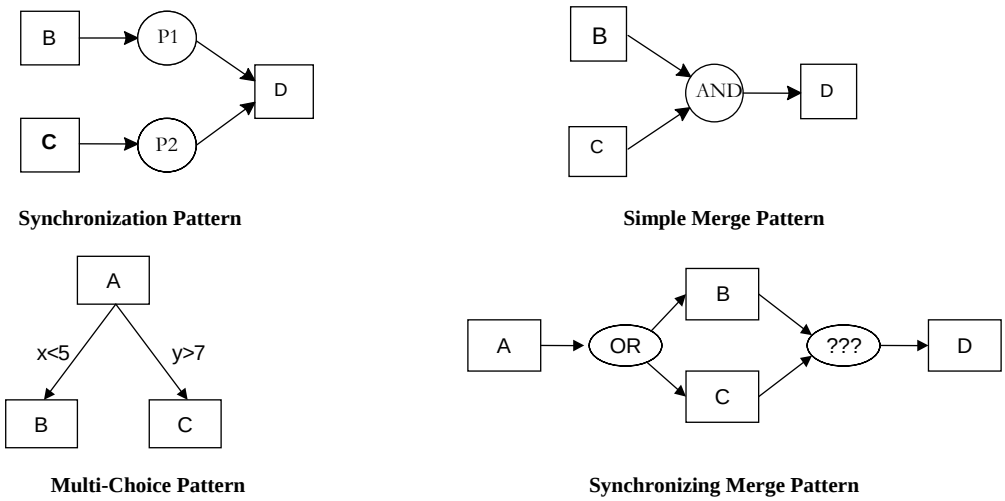


Figure I-7. Exemples de patrons de workflow

III.2.6. Autres catalogues de patrons de procédé

Outre les catalogues/langages de patrons de procédé présentés ci-dessus, il existe encore divers travaux concernant l'identification des patrons de procédé concrets. Nous citons ici les plus intéressants :

- **MSF patterns** : Dans [Pavlov04], les auteurs proposent des patrons de procédé et des patrons d'organisation extraits du Microsoft Solutions Framework (MSF). Les patrons représentés dans cet article sont *Living Document, Clonable Lifecycle, Smart Lifecycle* et *Stakeholder-Oriented Organization.*
- **Object-Oriented Reengineering Patterns** : Demeyer et al. [Demeyer02] proposent des patrons pour la rétro-ingénierie. Certains d'entre eux fournissent des démarches qui décrivent comment réaliser des solutions de patron.

- **Business Process Patterns** : La communauté Business Process a défini également des patrons de procédé qui permettent de décrire des procédés métier. Parmi les travaux les plus importants dans ce domaine, nous citons les patrons de procédé métier de Penker&Eriksson [Penker00] et le MIT Handbook [Malone03].
- **Organizational Patterns** : On peut trouver aussi des patrons de procédé dans certains systèmes de patrons d'organisation comme [Foote95][Coplien94][Harrison96] [Kerth95][Whitenack94].

III.2.7. Bilan

Les patrons d'OOSP, d'OPF et de Bergner sont des *patrons de procédé généraux* couvrant l'ensemble du cycle de développement du logiciel. Ils fournissent des collections de techniques, d'actions et/ou de tâches à suivre pour le développement de logiciels selon l'approche orientée-objet.

Les patrons de SIP par contre sont spécifiques d'un *domaine*. Dans ce catalogue de patrons, le principal objectif des patrons processus est de spécifier une démarche de développement des SIP facilitant l'usage des patrons produit.

Quant aux patrons de workflow, ils sont plus abstraits que les patrons de procédé abordés dans cette section. Un patron de workflow ne fournit pas un procédé ayant une sémantique spécifique pour résoudre un problème de développement, mais fournit une solution générique pour modéliser un enchaînement d'activités du procédé dans une situation récurrente. On peut trouver ce type de solution dans certains patrons d'organisation ou patrons de procédé métier, mais cette approche est peu abordée dans les patrons de procédé logiciels.

La plupart des patrons relatifs au procédé présentés ci-avant sont représentés de façon informelle. Cela ne permet pas une application directe, éventuellement automatique des patrons de procédé pour modéliser les procédés.

III.3. FORMALISATION DE PATRONS DE PROCÉDÉ

Comme nous l'avons souligné, la représentation informelle de patron de procédé empêche une exploitation efficace de cette notion dans la pratique. La formalisation de patrons de procédé est donc primordiale pour promouvoir leur application.

Le premier niveau de formalisation de patrons de procédé est la structuration de leurs contenus par un formalisme de représentation. Un tel formalisme propose une structure comportant un ensemble de rubriques caractérisant un patron. De nombreux formalismes ont été proposés dans la littérature pour représenter les patrons[1] [Ambler98][Conte01] [Storrle01][TranDT01]. Bien que les formalismes proposés diffèrent par le nombre et le degré de détail de leurs rubriques, ils intègrent tous le triplet essentiel {problème, contexte, solution} pour exprimer respectivement un certain problème récurrent, une solution pour ce problème et son contexte d'application.

[1] On n'aborde ici que les formalismes structurés et on ignore les formalismes narratifs tels que ceux d'Alexander [Alexander77] ou Portland [Portland00]

La Figure I-8 montre le formalisme proposé dans [Ambler98].

Nom :	nom qui décrit brièvement le patron
Type :	indique si le patron décrit est de type Tâche, Phase ou Etape.
Forces :	buts à atteindre et contraintes à respecter en appliquant ce patron.
Contexte initial :	conditions nécessaires pour déclencher l'application du patron.
Solution :	décrit les étapes ou les actions à exécuter pour résoudre le problème.
Contexte résultant:	conditions qui doivent être satisfaites après l'application de ce patron

Figure I-8. Formalisme de représentation de patrons de procédé [Ambler98]

La description informelle des patrons de procédé en langage naturel peut être un avantage, car elle ne demande aucune connaissance sur le formalisme de description pour comprendre un patron. Mais la langue naturelle peut produire des expressions ambiguës voire contradictoires, et la description informelle de procédés est inefficace si l'on veut l'utiliser comme une base de communication ou si l'on vise un support automatique par un outil de modélisation de procédés. C'est pourquoi le besoin d'un langage plus formel pour modéliser les patrons de procédé émerge de plus en plus.

Un tel langage doit fournir les concepts nécessaires pour décrire les patrons de procédé et préciser la sémantique des concepts qu'il véhicule pour limiter les ambiguïtés et les incompréhensions. Cependant, la formalisation de patrons de procédé a été peu abordée dans la littérature. A notre connaissance, il n'y a que quelques équipes qui travaillent sur ce sujet en utilisant la technique de la méta-modélisation pour définir un langage de description de procédés supportant le concept de patron de procédé. Nous introduisons dans la suite les travaux de ces équipes : le framework *Living Software Development Process* [Gnatz03], le langage de description des patrons *PROPEL* [Hagen04], le langage de description de procédés *PROMENADE*[Ribo01] et le méta-modèle de procédé logiciel *SPEM*[SPEM07].

III.3.1. Le framework Living Software Development Process

Le procédé *Living Process* est un procédé flexible de développement proposé par une équipe de recherche de l'université technique de Munchen [Gnatz03]. Ce procédé supporte l'évolution statique et dynamique pour s'adapter aux besoins des projets réels.

Pour permettre de modéliser les procédés, un méta-modèle de procédé a été proposé dans [Gnatz02]. Ce méta-modèle définit trois concepts principaux : *Work Artefact* décrivant les artéfacts produits ou requis par le procédé, *Process Artefact* décrivant les tâches de développement pour produire ou modifier les artéfacts, et *Context* définissant les interfaces reliant des *Process Artefacts* avec des *Work Artefacts* (Figure I-9).

Un procédé de développement et ses activités correspondantes sont décrits par des *Process Artefacts* qui sont spécialisés en *Process Patterns* et *Activity Descriptions*. Une *Activity Description* décrit l'interface d'une activité en spécialisant les *Work Artefacts* de l'activité, concrètement ses entrées et ses sorties, alors qu'un *Process Pattern* détermine comment réaliser une activité, concrètement ses sous-activités et l'enchaînement de ces activités. Cette séparation permet une flexibilité de description de procédés, car une activité peut être réalisée de plusieurs

manières en associant différents patrons de procédé. Un patron de procédé décrit donc un fragment de procédé réutilisable.

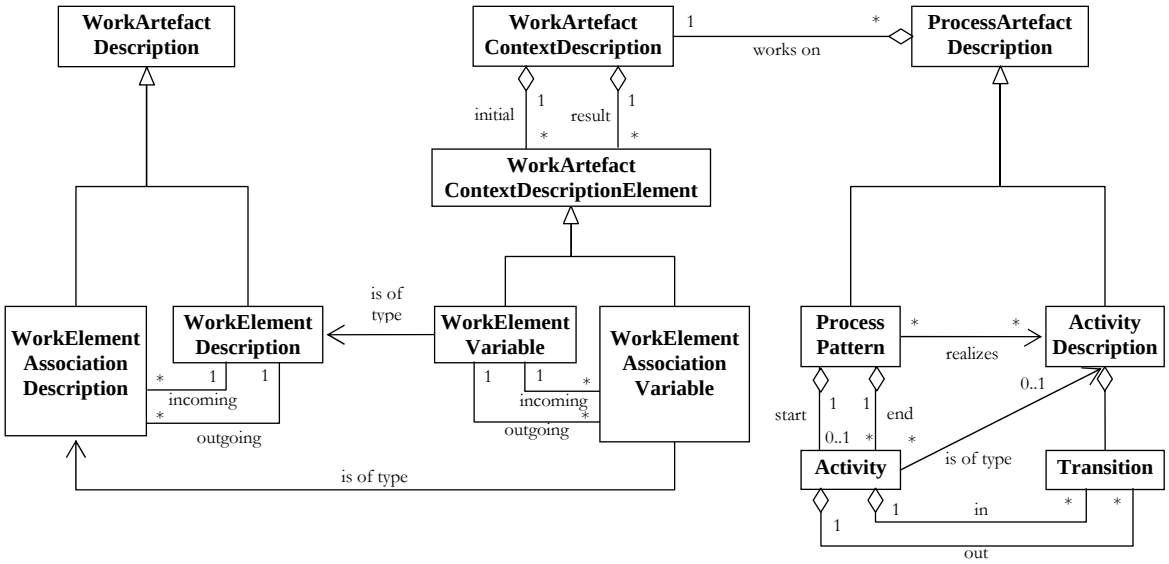


Figure I-9. Méta-modèle du framework Living Process [Gnatz03]

La Figure I-10 montre un exemple de patron de procédé décrit en utilisant le méta-modèle de Living Process.

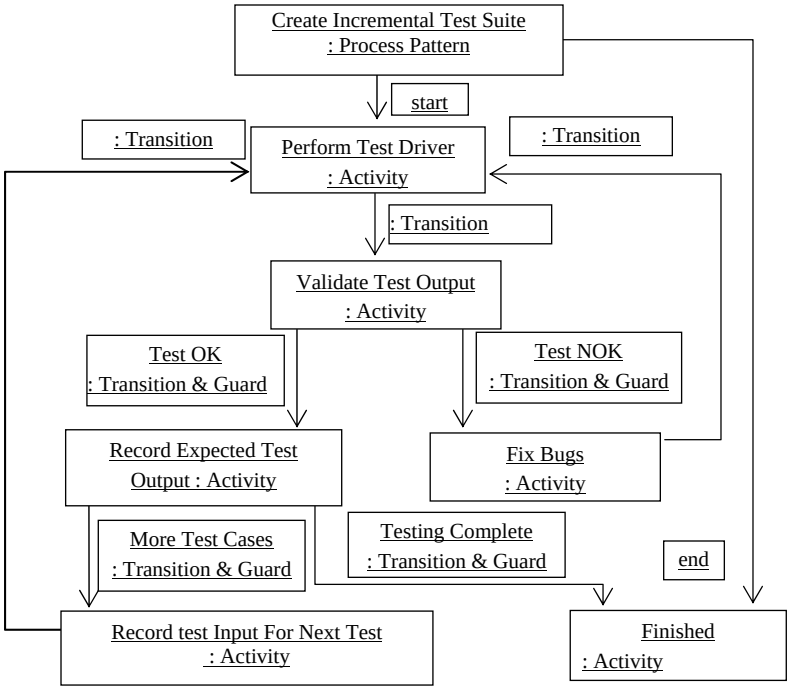


Figure I-10. Patron de procédé « Create incremental Test Suite » [Gnatz03]

Pour représenter un catalogue de patrons en reliant des patrons avec les activités qu'ils adressent, Gnatz et al. proposent la structure *ProcessPatternActivityMap* [Gnatz02] sous forme d'un graphe orienté (carte). Ce graphe est constitué de nœuds de type description d'activité ou

patron de procédé. Un noeud de type patron est relié par une relation *realizes* avec l'activité qu'il réalise, et par les relations *executes*[1] avec les activités qui doivent être exécutées en appliquant ce patron. Par conséquent, cette carte permet de montrer les solutions pour réaliser une activité. Nous montrons un exemple d'une telle carte dans la Figure I-11.

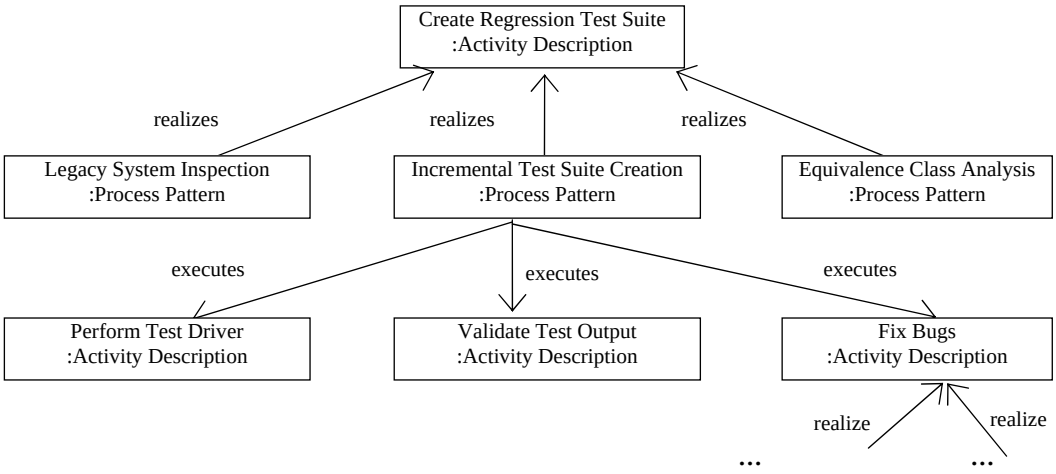


Figure I-11. Exemple de ProcessPatternActivityMap [Gnatz03]

III.3.2. Process Pattern Description Language (PPDL/PROPEL)

PROPEL est un langage de description de patron de procédé développé à l'université de Leipzig [Dittmann02][Hagen04]. Basé sur UML, ce langage permet de représenter les procédés de manière semi-formelle.

PROPEL a été développé spécialement pour représenter les patrons de procédé et leurs relations. Dans PROPEL, un patron de procédé (*ProcessPattern*) propose un procédé éprouvé (*Process*) pour résoudre un problème de développement logiciel récurrent (*Problem*) dans un contexte spécifique (*Context*). Un contexte définit les conditions qui doivent être vérifiées avant et après l'application du patron. Un patron ainsi qu'un problème sont caractérisés par un contexte d'entrée et un contexte de sortie. Il peut y avoir plusieurs patrons fournissant un ensemble de solutions pour un problème.

Un procédé dans PROPEL est modélisé comme une spécialisation du concept *ActivityGraph* de UML. Il décrit le déroulement d'un ensemble d'activités (*ActionState*) et le flux d'objets entre ces activités. Un rôle (*Role*) est responsable de l'exécution du procédé. Le procédé proposé par un patron contient plusieurs activités. A chaque activité correspond un problème (*ActivityProblemMapping*). Les problèmes assignés aux sous-activités d'une solution d'un patron sont des sous-problèmes du problème du patron. Par conséquent, il peut y avoir des patrons de procédé qui résolvent une étape partielle du patron supérieur.

La Figure I-12 montre un extrait du méta-modèle de PROPEL.

[1] Cette relation est déterminée par la relation "is of type " entre la description d'activité et les activités du patron de procédé assigné à cette description (cf. Figure I-9)

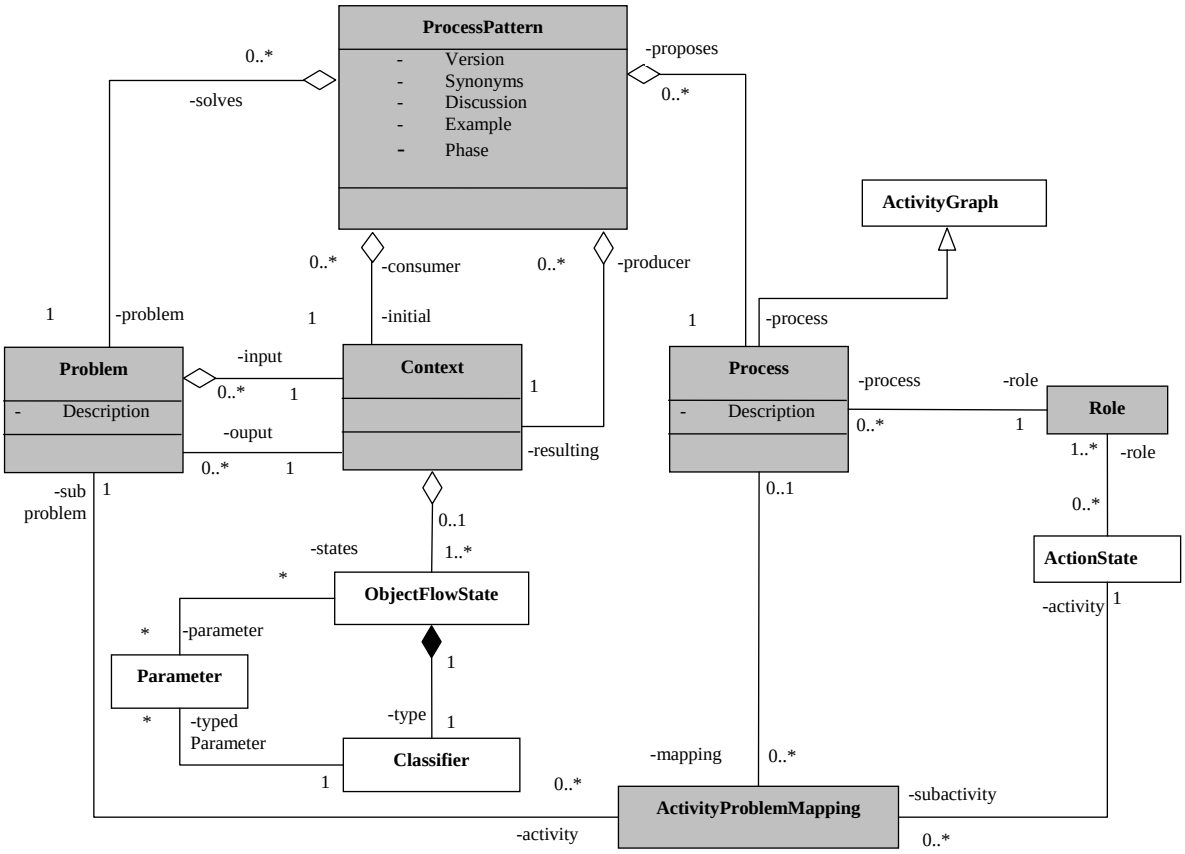


Figure I-12. Extrait du méta-modèle de PROPEL [Hagen04]

La Figure I-13 montre le procédé correspondant à la solution du patron «Review» représenté dans PROPEL. Ce procédé contient des activités ainsi que d'autres patrons.

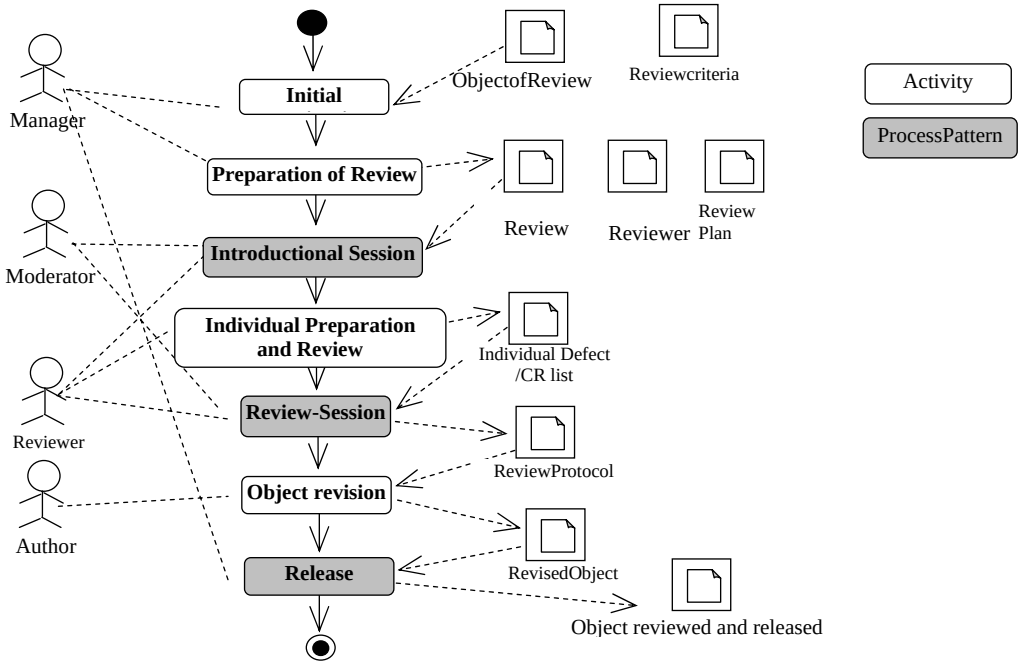


Figure I-13. Extrait du patron "Review" dans PROPEL [Dittmann02]

Pour faciliter l'organisation de patrons, PROPEL d'une part définit explicitement les relations entre patrons, d'autre part permet de regrouper plusieurs patrons en catalogues de patrons (*ProcessPatternCatalog*) (Figure I-14).

Grâce aux relations définies entre patrons, PROPEL permet de créer des diagrammes de patrons pour représenter un catalogue de patrons selon différents points de vue, chaque point de vue reflétant un type de relation entre patrons. Ceci aide les concepteurs à choisir les patrons nécessaires.

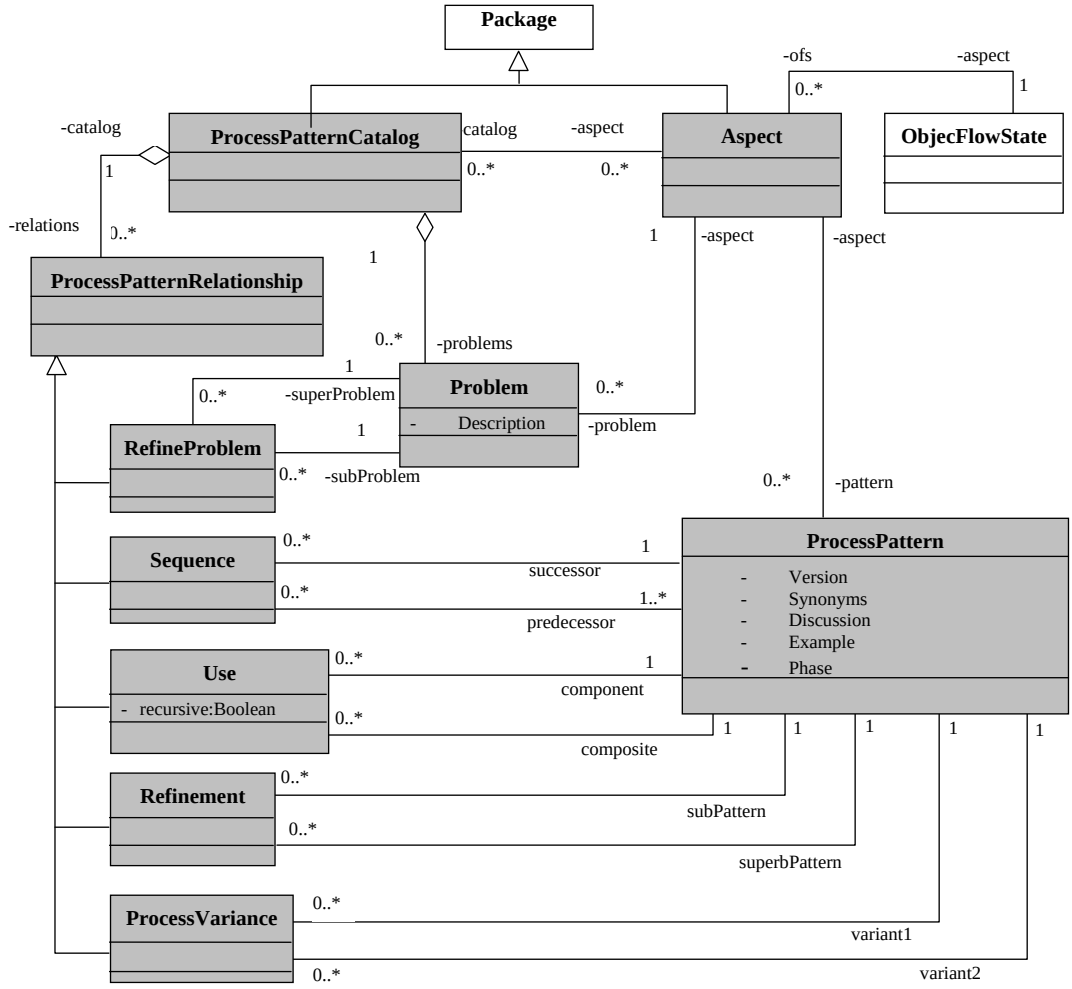


Figure I-14. Méta-modèle du paquetage Relationship de PROPEL [Hagen04]

III.3.3. PROMENADE

PROMENADE est un langage de description de procédés développé à l'Université polytechnique de Catalunya [Ribo00][Ribo01][Ribo02]. PROMENADE se base sur UML en ajoutant des extensions pour décrire les éléments de procédé, y compris les patrons de procédé.

Dans PROMENADE, un modèle de procédé comporte une partie statique décrivant la structure du procédé et une partie dynamique décrivant le comportement du procédé.

Pour décrire l'aspect statique des procédés, PROMENADE ajoute au méta-modèle d'UML les méta-classes *MetaTask*, *MetaDocument*, *MetaRole* représentant respectivement les

éléments de base activité, produit et rôle. La méta-classe *SPMetamod* représentant un modèle de procédé est un agrégat de ces éléments de base (Figure I-15).

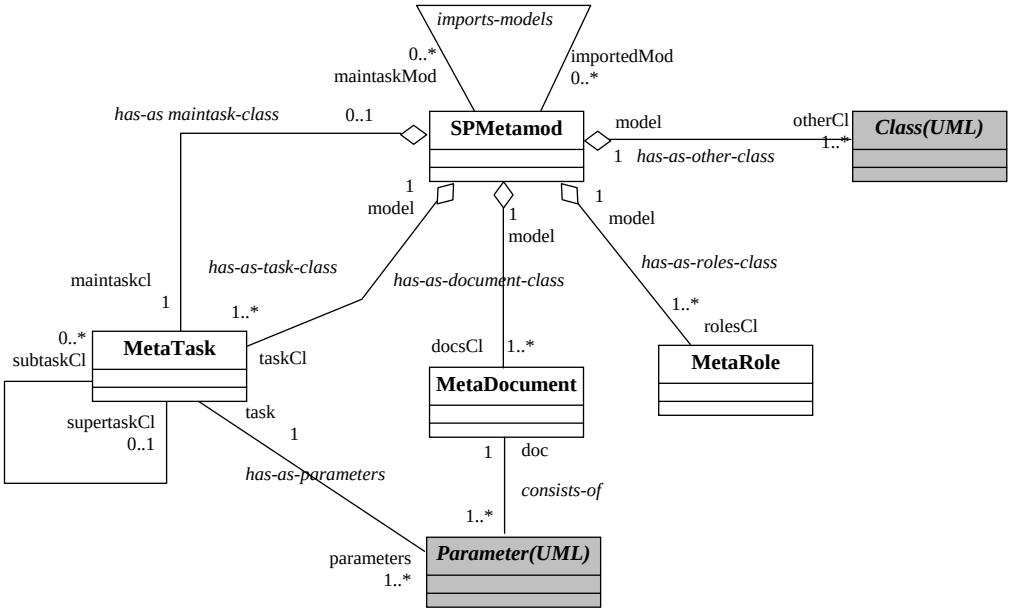


Figure I-15. Méta-modèle de procédé PROMENADE [Ribo00]

Dans PROMENADE un patron de procédé est défini comme une classe *SPMetamod* qui décrit un modèle de procédé offrant une solution pour un problème récurrent de développement. Ce concept est proposé avec l'intention de représenter les templates de modèle. Un patron de procédé peut donc peut être un modèle paramétré, ou un modèle contenant des éléments qui peuvent être raffinés plus tard. Ainsi, il permet de définir un modèle générique de procédé qui sera raffiné en l'instanciant avec des paramètres effectifs.

En particulier, PROMENADE définit explicitement la relation *PPBinding* pour spécifier l'application de patrons. De plus, PROMENADE permet la définition de contraintes spécifiques sur des paramètres de patrons de procédé grâce à la méta-classe *ParameterConstraint* (Figure I-16).

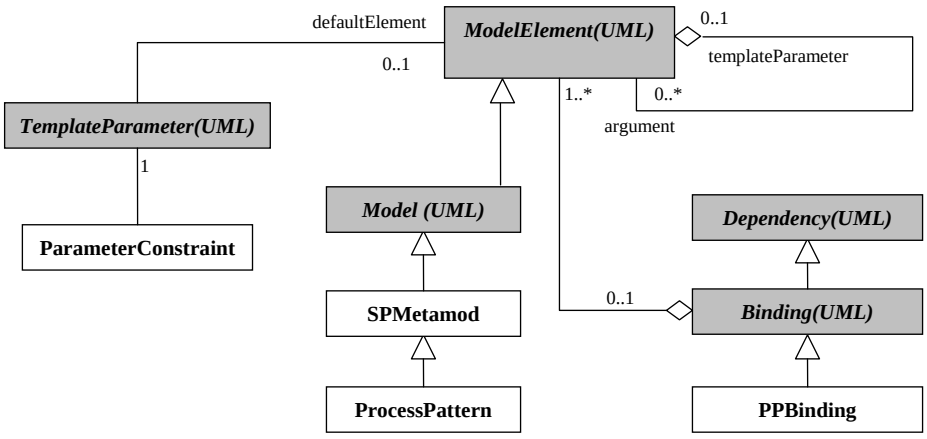


Figure I-16. Patron de procédé dans le méta-modèle de PROMENADE [Ribo02]

La partie dynamique du modèle de procédé dans PROMENADE décrit comment les activités d'un procédé peuvent être exécutées. PROMENADE fournit les deux paradigmes proactif et réactif pour spécifier le comportement des procédés. D'une part il établit les relations de précédence pour décrire le déroulement des activités ; d'autre part, il permet de définir des déclenchements et des exceptions pour intervenir à n'importe quel moment du déroulement prévu. La Figure I-17 montre l'extension du méta-modèle d'UML dans PROMENADE pour décrire les relations de précédence entre activités de procédé, et la Figure I-18 montre un exemple de patron de procédé PROMENADE.

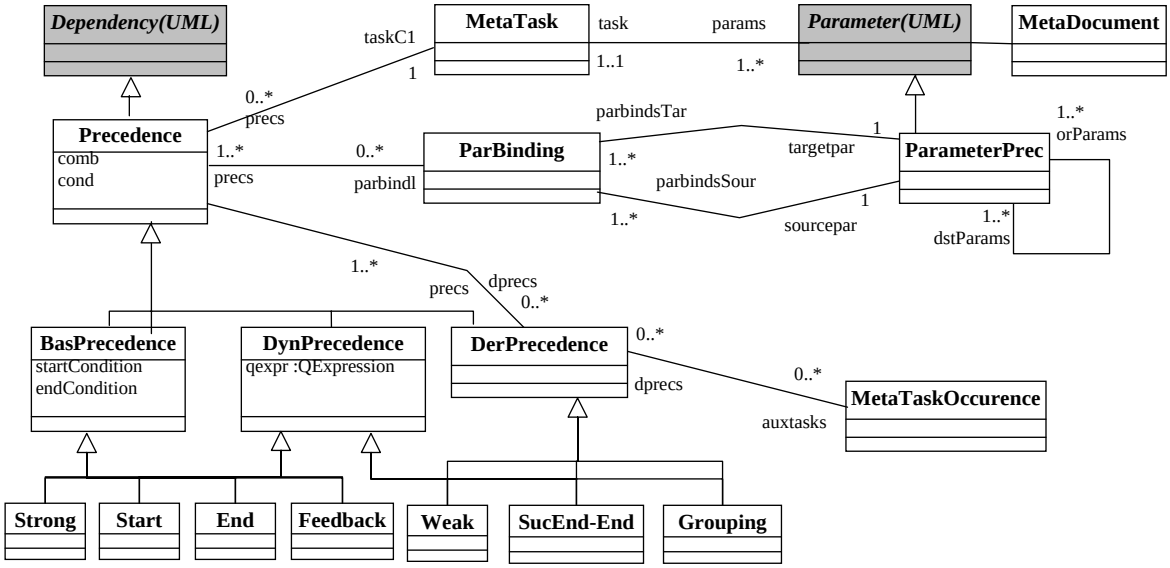


Figure I-17. Relations de précédence de PROMENADE [Ribo00]

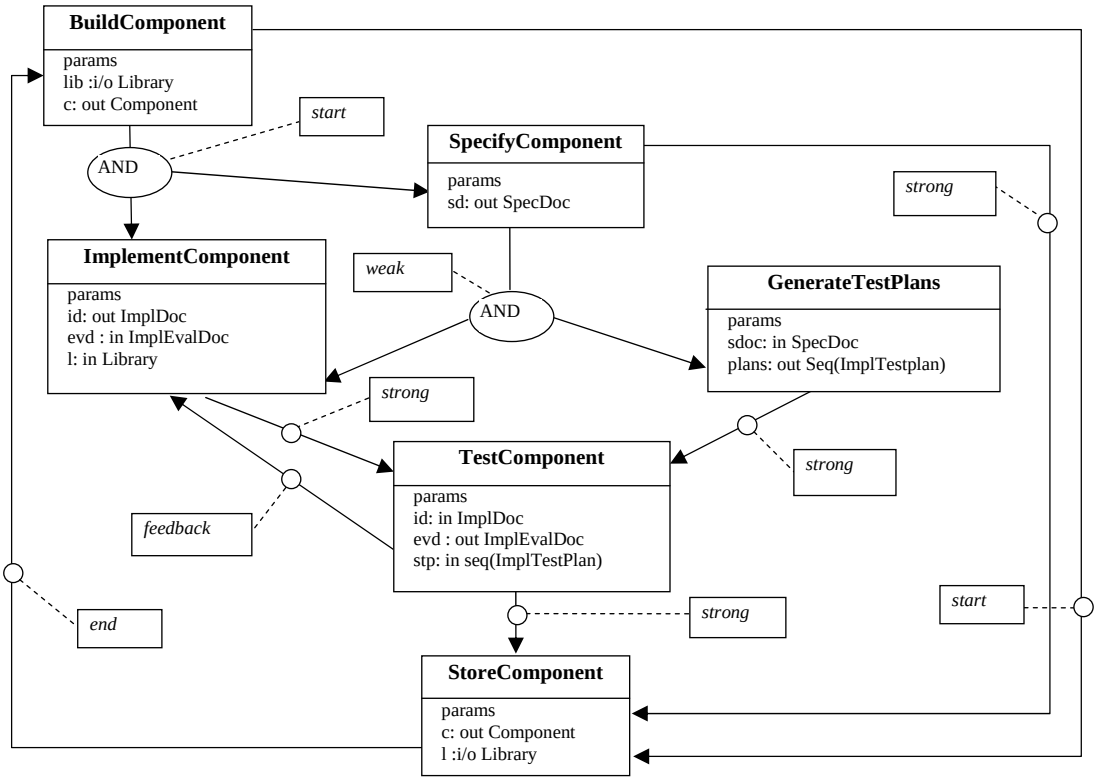


Figure I-18. Patron Build Component de PROMENADE [Ribo00]

III.3.4. Software Process Engineering Metamodel (SPEM)

SPEM est une norme de l'OMG dédiée à la description de procédés logiciels. SPEM propose un méta-modèle de langage de description de procédés basé sur UML. À ce jour, SPEM supporte la définition de modèles de procédé mais pas l'exécution de ces modèles.

Issu des travaux de l'OMG, SPEM s'inscrit doublement dans l'organisation de la pile de modélisation du MDA (Model Driven Architecture) : comme méta-modèle conforme au MOF (Meta Object Facility), et comme extension d'UML sous la forme d'un profil. Étant un méta-modèle MOF, SPEM peut étendre le méta-modèle UML en ajoutant librement les concepts dédiés aux procédés logiciels. Étant un profil UML, SPEM peut profiter des notations standardisées et des diagrammes expressifs de UML sans redéfinir la syntaxe concrète.

Le modèle conceptuel de SPEM repose sur l'idée qu'un procédé de développement de logiciel est une collaboration entre des entités actives et abstraites - les rôles - qui effectuent des opérations - les activités - sur des entités concrètes et réelles - les produits. Les différents rôles agissent les uns sur les autres ou collaborent en échangeant des produits et en déclenchant l'exécution de certaines activités (Figure I-19). Ce modèle conceptuel est la base du méta-modèle de SPEM.

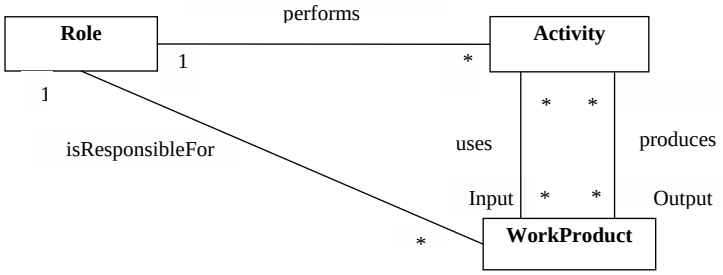


Figure I-19. Modèle conceptuel de SPEM

La spécification de SPEM actuellement disponible est la version 1.1 de 2005 [SPEM05] et fait suite à la toute première version publiée en 2002. Une proposition pour la version 2.0 de SPEM [SPEM07] est en cours d'étude. Nous nous intéressons aux deux versions de SPEM, car la version 1.1 offre la standardisation alors que la nouvelle version intègre le concept de patron de procédé.

SPEM 1.1

SPEM 1.1 est structurée en deux paquetages : *SPEM-Foundation* qui est une extension d'un sous-ensemble d'UML 1.4 fournissant les concepts de base pour définir le méta-modèle SPEM, et *SPEM-Extension* qui décrit des concepts de base pour décrire les procédés logiciels. La Figure I-20 montre le méta-modèle de SPEM1.1.

Le concept central de SPEM 1.1 est *WorkDefinition*, une unité de travail de procédé réalisée par un responsable (*ProcessPerformer)* pour produire ou modifier des artéfacts (*WorkProduct*).

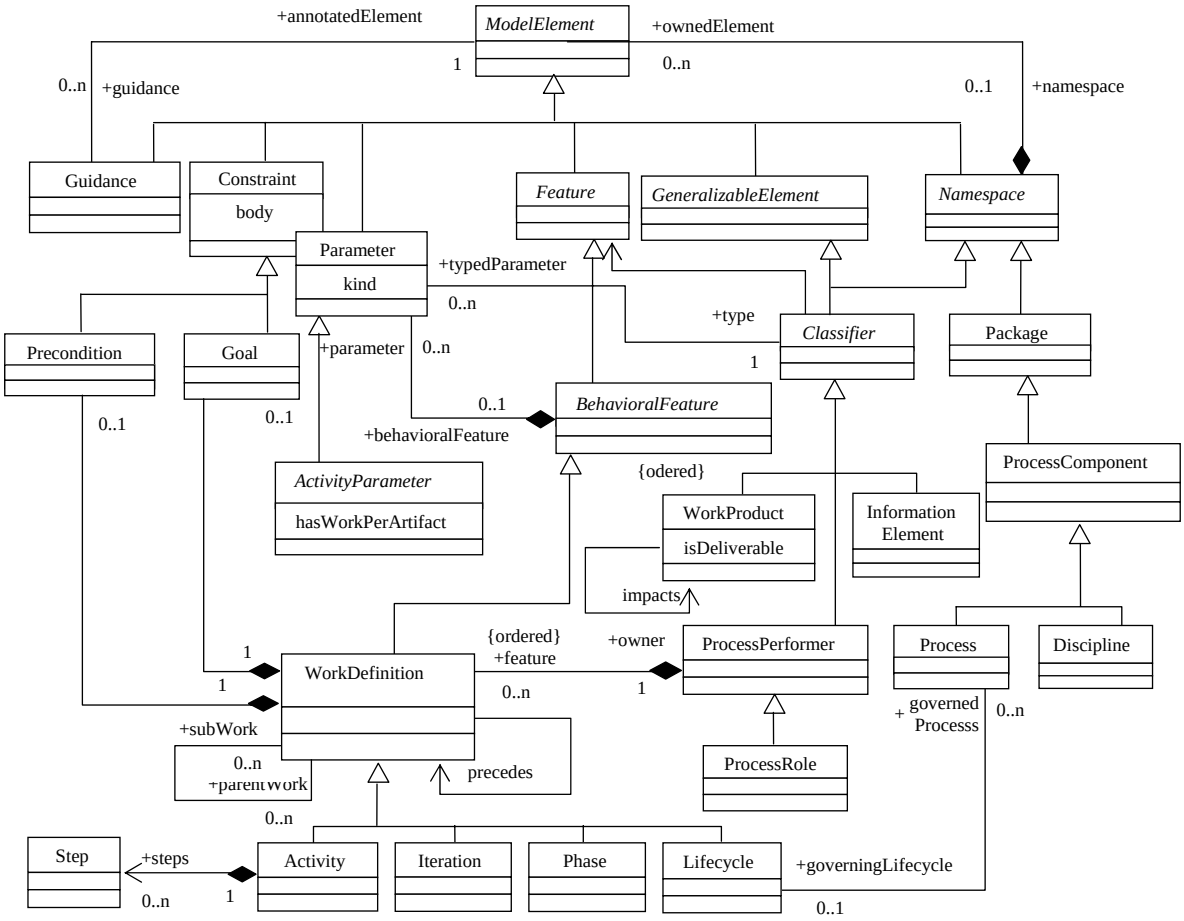


Figure I-20. Méta-modèle de SPEM1.1 [SPEM05]

L'exécution d'une *WorkDefinition* ne peut commencer que quand ses pré-conditions (*Precondition*) sont remplies ; cette exécution doit satisfaire des objectifs (*Goal)*, qui peuvent être vus comme des post-conditions de *WorkDefinition*. Ce concept est spécialisé en plusieurs sous-classes concrètes pour représenter des tâches de différentes granularités.

Une activité (*Activity*) est une spécialisation de WorkDefinition qui peut être assignée à un rôle individuel (*ProcessRole*)[1]. Une activité peut être décomposée en plusieurs étapes élémentaires (*Step)* définies dans le contexte de l'activité. Les autres spécialisations de WorkDefinition sont la phase (*Phase)*, l'itération (*Iteration*) et le cycle de vie (*Lifecycle*).

SPEM 1.1 supporte l'organisation modulaire et la réutilisation de procédés via le concept de composant de procédé (*ProcessComponent),* qui est spécialisé en discipline (*Discipline*) et procédé (*Process*). Une discipline regroupe des activités ayant le même thème. Un procédé est considéré comme un composant de procédé autonome contenant un ensemble de WorkDefinition et est gouverné par un cycle de vie.

[1] Ce concept se distingue de ProcessPerformer, une sorte de rôle abstrait responsable d'un ensemble de WorkDefinitions

SPEM 2.0

SPEM 1.1 a plusieurs points faibles, l'un d'eux étant l'insuffisance de flexibilité et de support de la réutilisation. Pour remédier à ces inconvénients, ainsi que pour aligner SPEM avec UML 2.0, l'appel pour une nouvelle version a été lancé en 2004 [SPEM04]. Nous présentons ici la proposition SPEM 2.0 d'IBM [SPEM07] en cours d'évaluation.

SPEM 2.0 applique le mécanisme de fusion de paquetage (*package merge*) de l'infrastructure d'UML 2.0 pour définir progressivement le méta-modèle. De plus, SPEM 2.0 fournit le mécanisme de «*plug-in*» pour faciliter l'extensibilité et la variabilité de procédés. Cela augmente considérablement la flexibilité de SPEM 2.0.

SPEM 2.0 sépare le contenu réutilisable d'une méthode de développement de son application dans les procédés. Le contenu d'un élément de méthode (*MethodContent*) fournit des explications décrivant la réalisation des buts spécifiques du développement de façon indépendante d'un cycle de vie concret. Un procédé, représenté par une activité (*Activity*), prend ces éléments de contenu de méthode et les relient selon l'ordre d'exécution adapté aux besoins d'un type de projet spécifique. Cela permet à SPEM 2.0 de mieux supporter la réutilisation de procédés.

SPEM 2.0 propose d'une part la notion de composant de procédé (*ProcessComponent*) remplaçable et réutilisable selon le principe de l'encapsulation (composant boite noire), d'autre part le concept de patron de procédé (*ProcessPattern*) pour la capitalisation des bonnes pratiques et l'assemblage rapide de nouveaux procédés. Nous nous intéressons en particulier au concept de patron de procédé de SPEM 2.0.

SPEM 2.0 ne définit pas de concepts spécialisés pour représenter les patrons de procédé. Un patron de procédé de SPEM 2.0 est simplement un procédé particulier qui décrit un ensemble d'activités réutilisables fournissant une solution de développement à un problème commun. Un patron est donc représenté comme un procédé ayant le type *ProcessPattern*.

La Figure I-21 montre un extrait du méta-modèle de SPEM 2.0 définissant le concept de patron de procédé.

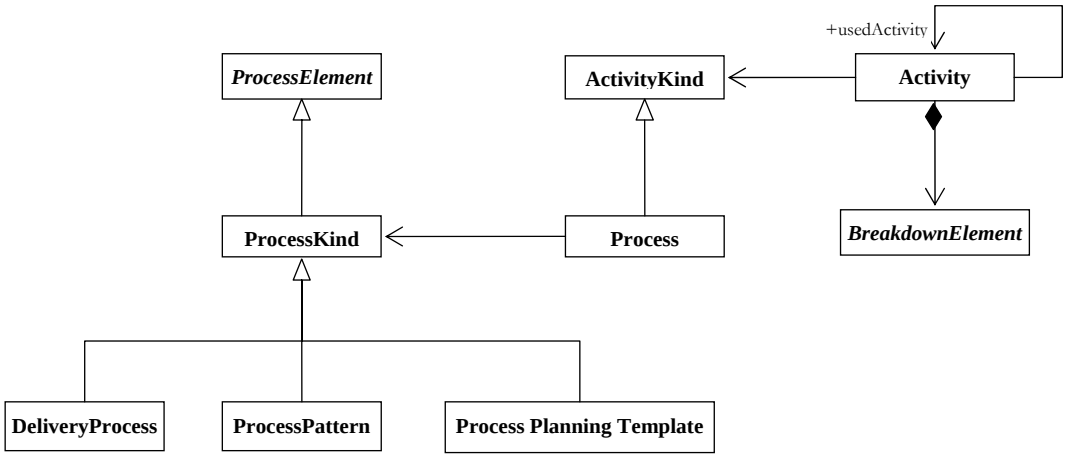


Figure I-21. Extrait du méta-modèle de SPEM 2.0 (paquetage *ProcessWithMethod*)

III.3.5. Bilan

Tous les formalismes présentés ci-dessus considèrent un patron de procédé comme un ensemble d'activités de développement réutilisables. Ils sont tous liés à UML mais à différents niveaux : PROPEL, PROMENADE et SPEM sont des extensions du méta-modèle UML ; LivingProcess Framework préconise simplement d'utiliser les notations UML pour représenter l'aspect dynamique des procédés.

Parmi ces formalismes, PROPEL est le seul dédié au concept de patron de procédé. Ses avantages sont la description de la structure interne d'un patron et la définition des relations entre patrons. Un des points forts de PROMENADE est de rendre les patrons de procédé plus génériques en les définissant en tant que modèles de procédé paramétrés. L'avantage de Living Process Framework est la spécification flexible de produits qui permet un assemblage possible de patrons hétérogènes.

L'avantage de SPEM 1.1 est sans conteste le méta-modèle standardisé dédié aux procédés logiciels. Cependant, SPEM 1.0 n'intègre pas le concept de patron de procédé. SPEM 2.0 propose un méta-modèle générique et flexible qui couvre divers types de procédés. Il ne met pas l'accent spécialement sur le concept de patron de procédé, mais fournit les concepts et les mécanismes généraux pour réutiliser les procédés en séparant les méthodes et leurs applications dans les procédés.

III.4. UTILISATION DE PATRONS DE PROCÉDÉ

Afin de pouvoir réutiliser les patrons de procédé d'une façon systématique, un *procédé par la réutilisation* est nécessaire. Ce procédé doit faciliter la sélection, l'adaptation et l'intégration de patrons de procédé pour modéliser les procédés logiciels. Autrement dit, il s'agit d'un méta-procédé fournissant une démarche et des mécanismes nécessaires pour aider les concepteurs de procédés à élaborer des modèles de procédé en réutilisant les patrons de procédé.

Dans la littérature dédiée au domaine de l'ingénierie des procédés, nous ne trouvons pas de tels méta-procédés complets. Nous présentons donc ci-dessous les travaux les plus proches de cette préoccupation, à savoir : le cadre des Compositional Patterns [Iida02], les mécanismes de réutilisation de modèles de procédé de PROMENADE [Ribo02], les mécanismes de réutilisation de méthodes de SPEM2.0 [SPEM07], et le noyau de méta-procédé du projet RHODES [Coulette02].

III.4.1. Compositional patterns

L'équipe d'Iida a proposé dans [Iida02] un cadre de "patrons compositionnels" pour modéliser les procédés logiciels. Cette approche est basée sur deux concepts :

- **Process Component** : c'est une unité de procédé qui peut être composée avec d'autres composants pour constituer un plus grand procédé. Un composant de procédé peut se configurer lui-même en se connectant avec les autres, et il est adaptable.
- **Composition Process Pattern** : c'est un patron qui représente la composition de procédé. Il définit un ensemble de composants de procédé et des liens entre eux. Un patron

compositionnel est utilisé comme un «template» de procédé (avec des éléments pouvant être remplacés par des sous-classes). Cette approche suppose que les composants de procédé soient capables de se connecter. Les patrons compositionnels peuvent être formellement décrits comme des connexions entre les composants de procédé.

La Figure I-22 montre les relations entre les deux concepts.

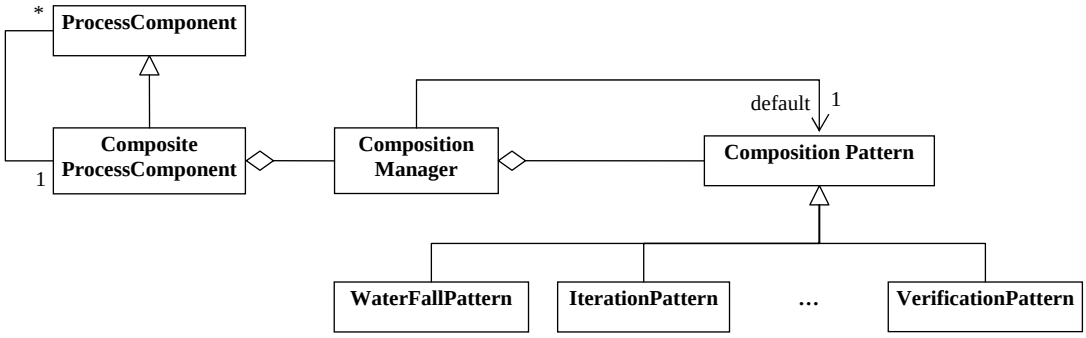


Figure I-22. Relations entre Process Component et Composition Pattern [Iida02]

Ce travail ne définit pas une démarche explicite de modélisation de procédés basée sur les patrons de procédé. Il propose simplement un guidage pour choisir des composants à composer et un patron de composition convenable pour composer ces composants. Il fournit aussi des règles d'application des patrons pour obtenir un composant composite. De plus un algorithme basé sur le parcours de graphe est proposé pour faciliter la recherche de patrons similaires.

III.4.2. Mécanismes de réutilisation de modèles de procédé dans PROMENADE

La réutilisation de procédés dans PROMENADE concerne deux activités générales et complémentaires : la *Récolte* (Harvesting) et la *Réutilisation* (Reuse). La *Récolte* est une transformation de modèles de procédé exécutables en patrons de procédé réutilisables. A l'opposé , la *Réutilisation* est une transformation de patrons de procédé en modèles de procédé exécutables.

Ces deux activités peuvent être effectuées en utilisant les mécanismes suivants :

- **Abstraction** : ce mécanisme permet d'éliminer les détails spécifiques d'un modèle de procédé pour obtenir un modèle plus générique. Deux mécanismes d'abstraction sont fournis dans PROMENADE : *généralisation* et *paramétrisation.*
- **Adaptation** : ce mécanisme a pour but de rendre un modèle de procédé (habituellement un patron de procédé) plus spécifique pour être réutilisé dans une situation particulière. Les mécanismes d'adaptation impliquent les modifications de modèles suivantes :
  - Addition de nouveaux éléments au modèle de procédé pour rendre le modèle de procédé plus spécifique : *spécialisation* et *instanciation*.
  - Suppression d'éléments (inutiles) du modèle de procédé pour limiter le modèle aux éléments choisis et à leur contexte : *projection*.

- Substitution d'éléments du modèle de procédé : *re-dénomination* et *substitution sémantique*.

- **Composition** : ce mécanisme combine un ensemble de modèles de procédé afin d'en créer un nouveau. PROMENADE distingue 3 mécanismes de composition :
  - *Grouping :* ce mécanisme regroupe un ensemble de modèles de procédé sans ajouter de sémantique supplémentaire spécifique.
  - *Combination :* ce mécanisme regroupe un ensemble de modèles de procédé en définissant plusieurs relations de précédence entre les tâches des modèles composants.
  - *Inclusion :* ce mécanisme incorpore la fonctionnalité d'un modèle de procédé à un autre modèle de procédé. Le modèle incorporé devient une sous-tâche d'une tâche complexe du modèle destinataire, avec éventuellement certains comportements supplémentaires.
- **Liaison tardive** : ce mécanisme permet de retarder le choix d'une partie spécifique d'un modèle de procédé jusqu' au moment de l'exécution.

La Figure I-23 montre la définition des opérateurs de réutilisation PROMENADE en tant qu'extension du méta-modèle UML.

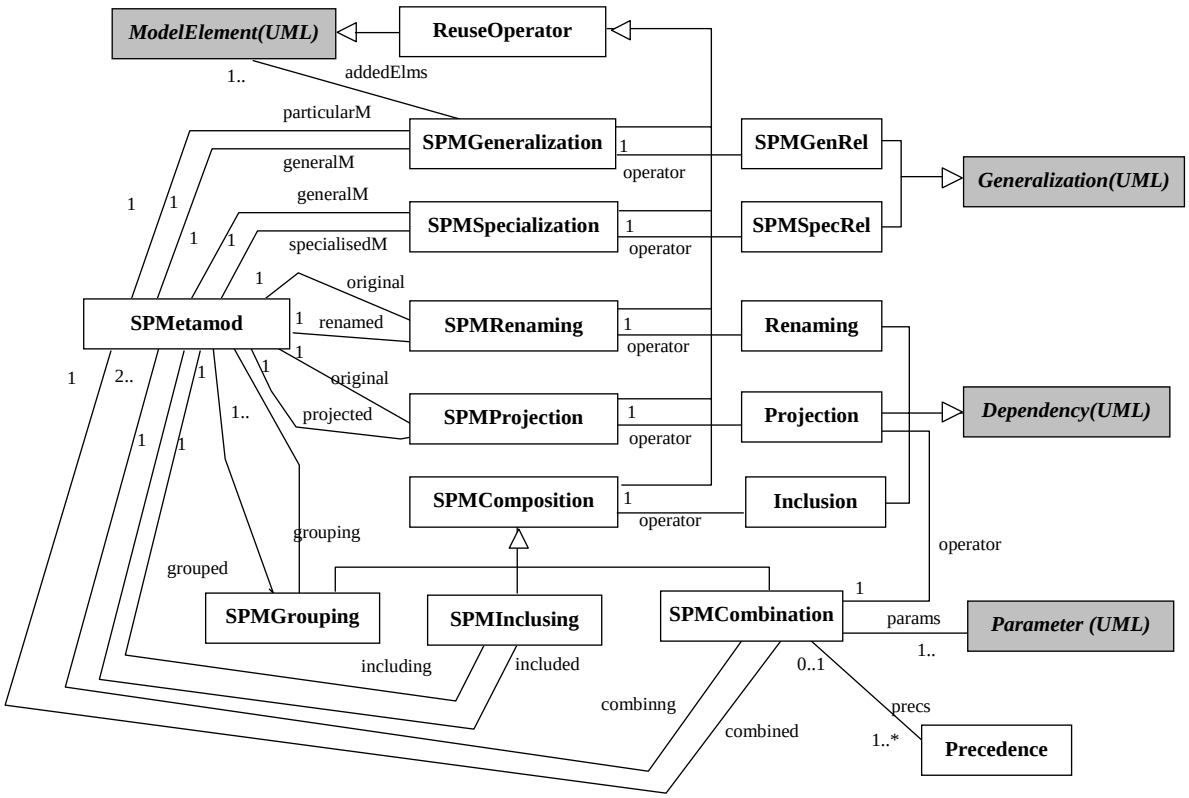


Figure I-23. Opérateurs de réutilisation de procédés dans PROMENADE [Ribo02]

PROMENADE fournit une définition spécifique de tous les mécanismes mentionnés ci-dessus. Une définition expressive des mécanismes de *paramétrisation* et *instanciation* est fournie dans le contexte des patrons de procédé (en fait, des patrons de procédé ont été définis en tant que modèles de procédé paramétrés (c.f. section III.3.3). Le reste des mécanismes ont été

définis au moyen d'un ou plusieurs opérateurs du langage PROMENADE. Ces opérateurs peuvent être appliqués aux modèles de procédé exécutables ou aux patrons de procédé.

III.4.3. Réutilisation de méthodes de développement en SPEM 2.0

La réutilisation des éléments de méthode dans SPEM 2.0 est réalisée grâce au concept de *Method Content Use* (Figure I-24).

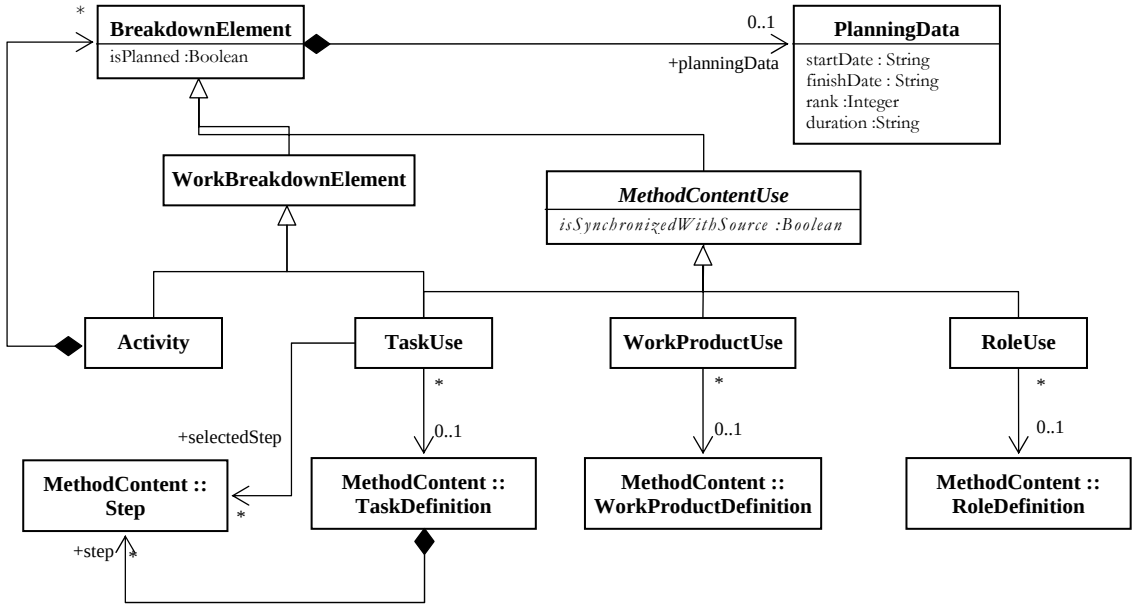


Figure I-24. Extrait du méta-modèle de SPEM 2.0 définissant le concept de Method Content Use

Un *Methode Content Use* fournit une représentation intermédiaire d'un *Method Content Element* dans le contexte d'une activité spécifique. Étant un *BreakdownElement*, il peut être contenu dans une activité. Il ne définit pas le contenu de l'élément mais référence un *Method Content Element* concret fournissant la définition pour l'élément dans le contexte de ce procédé. Ainsi, un *Method Content Element* peut être représenté par différents *Méthode Content Use* chacun dans le contexte d'une activité avec son propre ensemble de relations avec d'autres éléments de cette activité. Par conséquent, on peut réutiliser un *Method Content Element* pour définir plusieurs éléments de procédé.

La Figure I-25 montre un exemple de réutilisation d'éléments de méthode (en gris) par plusieurs *Method Content Use*. Par exemple, les deux éléments *Prioritize Usecase in Inception* et *Prioritize Usecase in Elaboration* sont deux *Task Use* référençant un même élément de méthode, i.e. la définition de tâche *Prioritize Usecase*. La définition de cette tâche spécifie le rôle participant (*Software Architect*) et les entrées, sorties (*Use Case Model, Requirement Attributes*). Ces éléments sont aussi réutilisés par les *Role Use* et *Work Product Use* correspondants. En réutilisant cette définition, chaque *Task Use* peut adapter ou ajouter ses éléments selon son contexte. Par exemple, *Prioritize Usecase in Elaboration* a deux sorties alors que *Prioritize Usecase in Inception* n'en a qu'une, comme la tâche *Prioritize Usecases*.

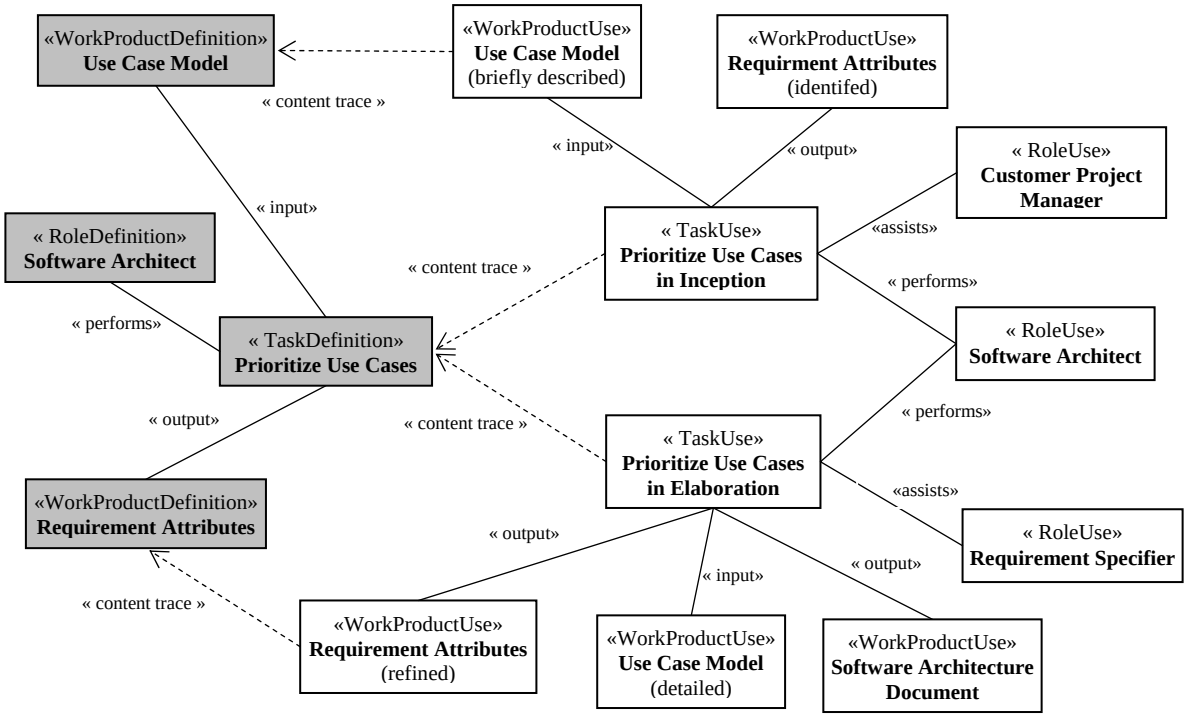


Figure I-25. Exemple d'utilisation en SPEM 2.0 des *Method Content Use* pour référencer des éléments de méthode.

La réutilisation de patrons peut être réalisée de deux façons : opérations de type *copier-coller* pour permettre aux concepteurs d'adapter les patrons selon leurs besoins ; établissement d'une relation entre une activité à définir et le patron à réutiliser (aussi une activité) en spécifiant le type d'application (attribut useKind) via des valeurs prédéfinies (ActivityUseKind) (Figure I-26)

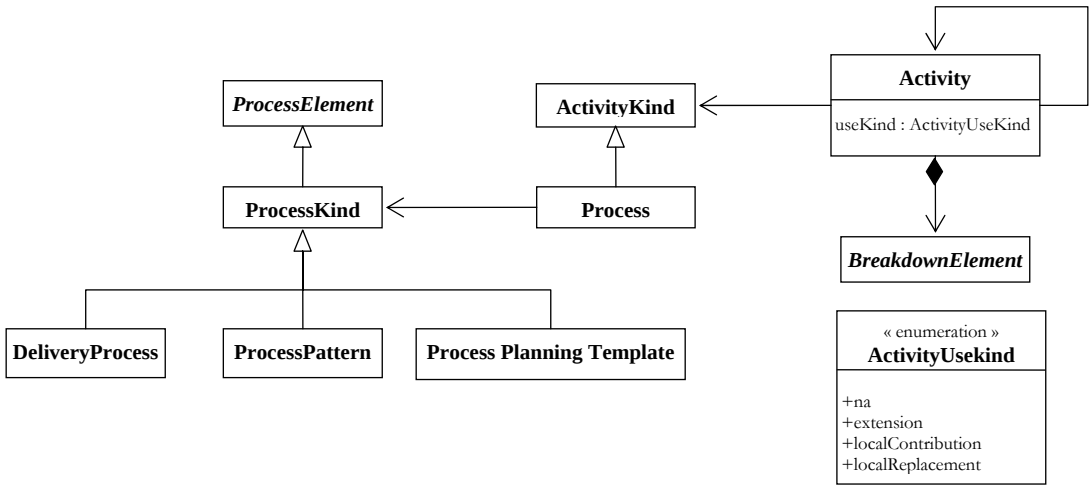


Figure I-26. Réutilisation de patrons de procédé par ActivityUse

La Figure I-27 montre un exemple représentant le procédé RUP constitué en réutilisant quatre *ProcessPatterns* (en gris). Un patron peut aussi être assemblé en utilisant d'autres patrons (ici le patron décrivant la conception de la base de données relationnelle (RDBMS) a réutilisé le patron OOAD).

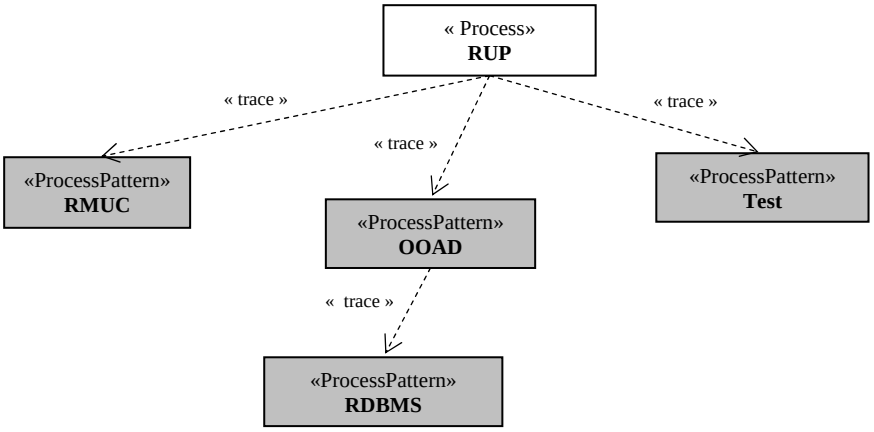


Figure I-27. Exemple de réutilisation de Process Patterns dans SPEM 2.0

III.4.4. Le méta-procédé de RHODES

L'environnement RHODES [Coulette00] est un atelier de génie logiciel centré procédé (AGL-P) qui a été développé à l'IRIT-ENSEEIHT. La description d'un procédé se fait à l'aide du LDP dédié orienté objet et activité PBOOL [Crégut97][TranDT01]. Le langage PBOOL supporte le non déterminisme à travers la notion de schéma de réalisation d'une activité. Un procédé PBOOL peut être exécuté sous RHODES pour contrôler le développement et assister les développeurs.

Pour rendre la réutilisation de processus efficace, RHODES utilise l'approche fondée sur les composants de procédé [Coulette01] et propose un noyau de méta-procédé pour construire et réutiliser des procédés logiciels [Coulette02].

Un composant de procédé de RHODES est défini comme une unité cohérente et réutilisable de procédé. Deux niveaux de composants de procédé sont mis en évidence : les *composants élémentaires* de procédé décrivant les entités de base d'un procédé (activité, produit, rôle), les *composants complexes* constitués de composant élémentaires permettant de mieux structurer et réutiliser des éléments de procédé. Un composant complexe possède une spécification et peut posséder plusieurs implantations.

Le méta-procédé de RHODES permet de passer progressivement d'un procédé informel à un procédé décrit sous forme de composants PBOOL. Il prend en compte à la fois la gestion globale d'un procédé et la description de composants de procédés de granularité fine. Ce méta-procédé supporte l'évolution et la réutilisation d'entités élaborées, et il est suffisamment formel pour permettre une assistance fine aux concepteurs de procédés.

Plus précisément, le méta-procédé RHODES comprend trois cycles itératifs : un *cycle général de modélisation*, un *cycle de formalisation d'un composant de procédé*, et un *cycle de recherche et de réutilisation d'un composant de procédé*. Le cycle général est utilisé pour définir un procédé. Les deux autres cycles sont des cycles de raffinage. Chaque cycle du méta-procédé est intercalé itérativement dans une étape ou une activité du cycle précédent. Le cycle général peut être instancié lui-même lors de sa décomposition (Figure I-28).

La Figure I-29 montre les détails du cycle général de modélisation. Dans ce cycle, le procédé considéré peut, selon sa complexité, être modélisé directement (suivant le schéma *Établir la conception*) ou décomposé en sous-procédés (suivant le schéma *Décomposer*). Le schéma *Établir la conception* comporte quatre étapes : décrire informellement les composants participant au procédé (*Description informelle*) ; établir une conception semi-formelle du procédé (*Description semi-formelle*) ; formaliser le procédé en exécutant les activités du *Cycle de formalisation* (*Formalisation*), ce qui donne lieu à un ensemble de composants réutilisables décrits en PBOOL ; valider la description formelle PBOOL (*Validation*).

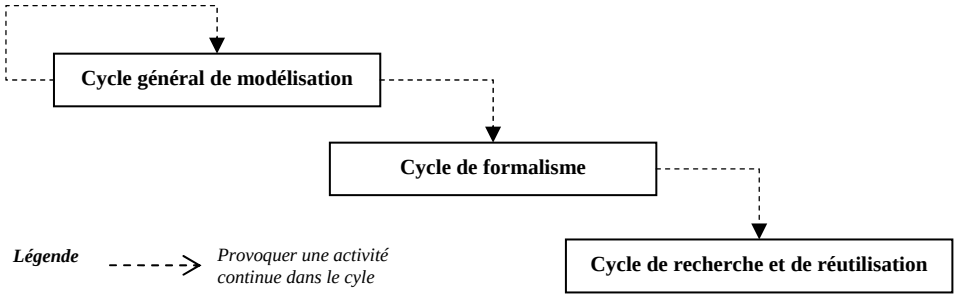


Figure I-28. Méta-procédé RHODES [TranDT01]

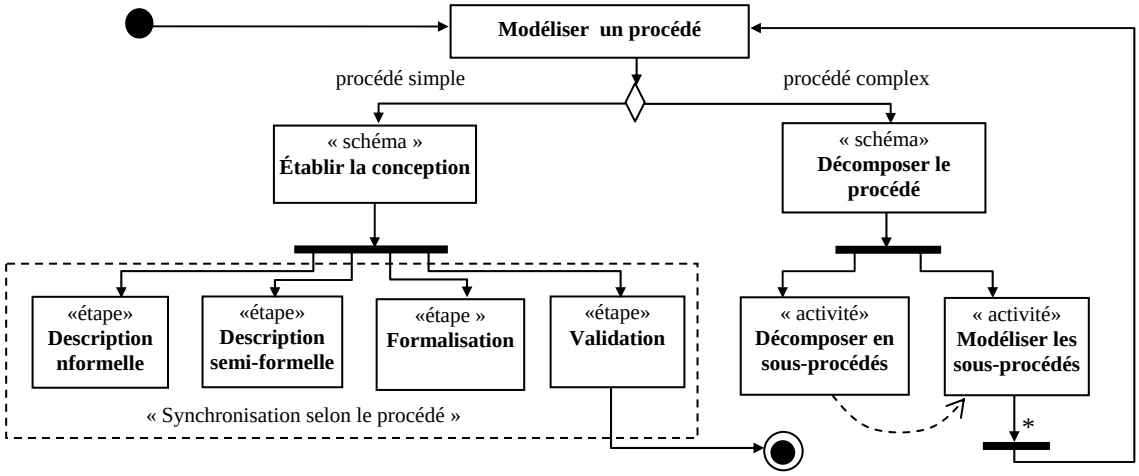


Figure I-29. Cycle général de la démarche de modélisation d'un procédé [TranDT01]

Le cycle de recherche et de réutilisation d'un composant de procédé est employé pour retrouver les composants dont les spécifications sont conformes à une spécification donnée, et permettre de choisir un composant convenable pour le réutiliser ou l'adapter. Ce cycle est décomposé en trois activités : une activité *S* pour rechercher des composants convenables, l'activité *Choisir une entité* pour choisir le composant le plus approprié, et l'activité *Réutiliser une entité* pour réutiliser le composant choisi (Figure I-30).

Pour permettre une spécification dynamique des critères de recherche, l'activité S est représentée sous forme d'un paramètre formel de l'activité racine du cycle. L'activité *Réutiliser une entité* propose trois schémas de réalisation qui permettent de dériver une nouvelle entité à partir d'un composant existant, d'en créer une nouvelle en adaptant une existante, ou d'instancier une entité existante.

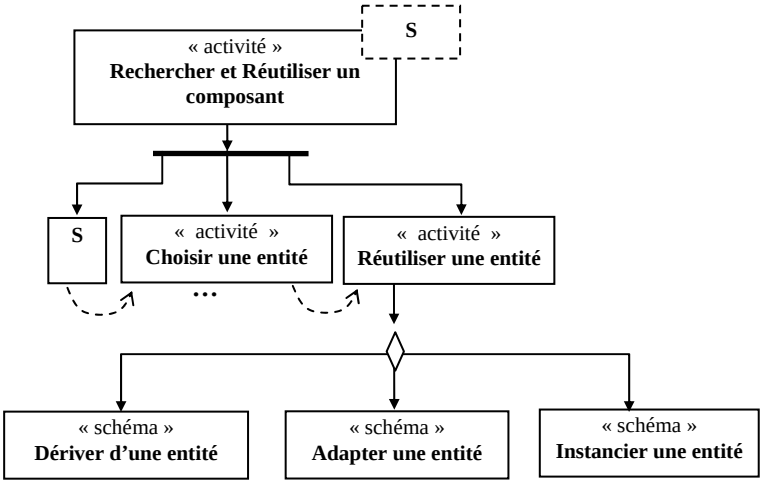


Figure I-30. Cycle de recherche et de réutilisation d'un composant

III.4.5. Bilan

Comme nous l'avons remarqué ci-avant, il y a peu de travaux dédiés à l'utilisation (ou réutilisation) de patrons de procédé pour modéliser les procédés.

Aucune approche existante ne satisfait complètement les besoins attendus d'un méta-procédé favorisant la réutilisation de patrons. PROMENADE fournit un ensemble d'opérateurs bien définis pour manipuler les modèles, mais il manque un guidage méthodologique pour une réutilisation systématique. SPEM 2.0 propose un mécanisme général qui permet une réutilisation assez flexible de procédé. Cependant, l'adaptation des éléments réutilisables fournie par SPEM2.0 est encore limitée et informellement définie. RHODES propose un méta-procédé guidant l'élaboration de modèles de procédé en réutilisant des composants de procédés, mais ce méta-procédé reste à un niveau de description de type gros-grain, et surtout il ne traite pas la notion de patron en tant que telle.

III.5. SYNTHÈSE

Bien que le concept de patron de procédé soit prometteur, les travaux dans ce domaine sont encore modestes [SDPP02]. La majeure partie des travaux existants se concentre sur l'identification des patrons spécifiques. En outre, la plupart des systèmes de patrons de procédé sont présentés informellement. Les études sur la formalisation de patrons de procédé sont encore limitées et les formalismes proposés ne sont pas encore acceptés largement par la communauté des informaticiens, notamment dans l'industrie. Il n'y a quasiment aucune recherche significative sur des méthodologies permettant de réutiliser des patrons de procédé.

Pour faciliter la comparaison des travaux sur les patrons de procédé présentés ci-dessus, nous montrons dans le Tableau I-4 et le Tableau I-5 une évaluation de ces travaux en utilisant le cadre de référence proposé dans la section II.3.

Besoin	Facette	Attribut	Living Process Framework	PROPEL	PROMENADE	SPEM (1.1 & 2.0)	RHODES
Formalisation de connaissances réutilisables	Identification des connaissances	Nature	Connaissance de Développement	Connaissance de Développement	Connaissance de Développement	Connaissance de Développement	Connaissance de Développement
		Couverture	Dépendante d'un domaine	Dépendante d'un domaine	Dépendante d'un domaine	Dépendante d'un projet et domaine	Dépendante d'un domaine
	Représentation des connaissances	Expressivité	Éléments de Procédé ; Éléments réutilisables	Éléments de Procédé[1] ; Éléments réutilisables	Éléments de Procédé ; Éléments réutilisables	Éléments de Procédé ; Éléments réutilisables	Éléments de Procédé ; Éléments réutilisables
		Modularité	Supportée	Supportée	Supportée	Supportée	Supportée
		Abstraction	Un niveau	Un niveau	Plusieurs niveaux [2]	Un niveau	Un niveau[3]
		Standardisation	Non-standardisée	Non-standardisée[4]	Non-standardisée[5]	Standardisée	Non-standardisée
		Formalisation	Semi-formelle	Semi-formelle	Formelle	Semi-formelle	Formelle
	Organisation des connaissances	Encapsulation	Patron de Procédé	Patron de Procédé	Patron de Procédé	Composant de Procédé; Patron de Procédé[6]	Composant de Procédé[7]
		Structuration	Carte de connaissance	Carte de connaissance	Bibliothèque de connaissance	Bibliothèque de connaissance	Bibliothèque de connaissance

Tableau I-4. Évaluation des travaux sur la formalisation de patrons de procédé

[1] PROPEL met l'accent sur la représentation du concept de patron de procédé et néglige la représentation d'éléments de procédé.

[2] PROMENADE support deux niveaux d'abstraction : les modèles de procédé et les templates de modèles (modèles paramétrés).

[3] RHODES introduit le concept de composant paramétré pour augmenter l'abstraction de modèles, mais ce concept reste informel dans le LDP PBOOL.

[4] PROPEL utilise un sous-ensemble de concepts et de notations d'UML, cependant, la définition de concepts dédiés au procédé et au patron est spécifique.

[5] PROMENADE utilise la même approche que PROPEL en étant fondé sur UML, mais développe ses propres concepts et notations pour décrire les procédé et les patrons.

[6] Seulement SPEM 2.0 définit le concept de patron de procédé.

[7] RHODES aborde le concept de patron de procédé mais ne le formalise pas.

Besoin	Facette	Attribut	Living Process Framework	PROPEL	PROMENADE	SPEM (1.1 & 2.0)	RHODES
Application de connaissances réutilisables	Opérateurs de manipulation	Mécanisme	Spécialisation	Spécialisation	Spécialisation ; Paramétrisation; Composition	Spécialisation; Composition	Spécialisation; Composition
		Définition	Implicite	Implicite	Explicite	Implicite	Implicite
	Guidage méthodologique	Type de guidage	ND	ND	ND	Conseil de réutilisation	Démarche de modélisation
		Spécification	ND	ND	ND	Gros-grains	Gros-grains
	Outils support	Catégorie	Gestion de bibliothèque de composants + Environnement de modélisation[1]	Gestion de bibliothèque de composants[2]	Environnement de modélisation[3]	Environnement de modélisation[4]	Environnement de modélisation[5]
		Mode de support	Semi-automatique	Semi-automatique	Semi-automatique	Semi-automatique	Semi-automatique

Tableau I-5. Évaluation des travaux sur l'utilisation de patrons de procédé

Note :

– ND signifie «Non Disponible». Cette valeur est utilisée dans les cas où un travail ne supporte pas un attribut de facette. .

[1] Dans le cadre du projet LivingProcessFramework il y a eu deux prototypes de gestion de connaissances de procédé [LISA02] et d'application de patrons [APE02] qui ont été développés séparément.

[2] PROPEL dispose d'un outil de gestion des patrons de procédé supportant leur réutilisation dans la modélisation de procédés. Cet outil est un prototype développé dans le cadre du travail de [Schröder03].

[3] PROMENADE possède un environnement supportant la modélisation de procédés en mettant l'accent sur le moteur d'exécution.

[4] SPEM 2.0 est supporté par l'outil gratuit du projet EPF [EPF06] proposé par IBM.

[5] L'environnement RHODES+ [TranDT01] se focalise sur la gestion de composants de procédé, il ne supporte pas la réutilisation de patrons de procédé.

IV. CONCLUSION

Dans ce chapitre, nous avons montré dans un premier temps l'importance de la modélisation et de la réutilisation de procédés logiciels. Nous avons vu ensuite le rôle du concept de patron de procédé dans le contexte de la réutilisation de procédés en indiquant ses intérêts supposés ainsi que les besoins auxquels il répond.

Dans l'étude bibliographique, nous avons établi un large panorama des travaux existant dans la modélisation de procédés et en particulier dans la formalisation et la réutilisation de patrons pour modéliser les procédés.

Ce travail a permis de mettre en évidence un certain nombre de points à approfondir concernant la notion de patron de procédé, ce qui explique pourquoi cette approche prometteuse reste encore peu exploitée en pratique.

Limites des travaux existants

Nous listons ci-dessous ce qui nous semble être les principales limites des travaux existants sur les patrons de procédé :

- **Définition vague et limitée du concept de patron de procédé**

Le concept de patron de procédé est actuellement utilisé avec une confusion conceptuelle et terminologique. Outre les phénomènes de synonymie et d'homonymie sur le terme de «patron de procédé» lui-même, il existe peu de définitions précises de la notion de patron de procédé.

Généralement, les patrons de procédé sont considérés comme des patrons capturant des activités réutilisables de développement, et utilisés comme des modules de base pour construire de nouveaux procédés. Nous pensons que cette définition n'est pas adéquate pour représenter l'idée originelle de patron de procédé qui vise à capturer et réutiliser diverses connaissances sur les procédés.

- **Formalisation insuffisante de patrons de procédé**

Comme nous l'avons expliqué ci-dessus, la représentation informelle de patrons de procédé ne permet pas leur application directe dans la modélisation de procédés. Pour rendre le concept de patron de procédé utilisable en pratique, les travaux présentés dans la section III.3 se concentrent sur la formalisation de patrons de procédé. Cependant, ils se bornent à formaliser des activités de développement réutilisables, et ne tiennent pas compte des autres types de connaissance plus abstraite sur la construction de procédé.

Dans le même sens, l'application de patrons de procédé est insuffisamment définie dans les formalismes pour permettre leur exploitation efficace dans la modélisation de procédés. Living Process Framework et PROPEL proposent une application simpliste des patrons ; PROMENADE et SPEM fournissent des mécanismes plus sophistiqués pour permettre l'adaptation de patrons au cours de leur application. Cependant, ces travaux ne proposent qu'un seul type d'application de patron : en considérant le patron comme un module de base pour construire de nouveaux procédés.

Vu les différents types de connaissance de procédé pouvant être capturés dans les patrons, nous pensons que les patrons de procédé doivent servir également comme structures (*templates)* pour restructurer ou enrichir les procédés dans le but de les améliorer[1]. Ce type d'application de patrons de procédé est encore négligé par la communauté des procédés.

Un autre problème relatif aux formalismes existants est qu'ils ne font pas l'objet d'un consensus sur les concepts de modélisation de procédés. Chaque formalisme définit des concepts différents, bien que souvent très proches. Cette diversité nuît d'une certaine manière au partage et à l'échange de modèles de procédés, ce qui est un frein évident à la réutilisation de procédés.

- **Réutilisation manuelle et ad-hoc de patrons de procédé**

Si les travaux sur la formalisation de patrons de procédé sont limités, il y en a encore moins sur l'aspect méthodologique de la réutilisation de patrons dans la modélisation de procédés.

La plupart des travaux présentés dans ce chapitre fournissent un formalisme semi-formel pour représenter les patrons de procédé, mais pas ou peu d'opérateurs pour les manipuler et les appliquer de façon automatique ou tout au moins assistée. PROMENADE est la seule approche qui définit une sémantique rigoureuse pour les opérateurs de manipulation et d'application de patrons.

Au niveau méthodologique, il manque une méthode pour guider l'application systématique de patrons dans la modélisation de procédés. Parmi les travaux représentés, seul RHODES propose une démarche pour guider la modélisation de procédés à base de composants réutilisables. Cependant, le méta-procédé de RHODES est défini d'une façon schématique et ne traite pas les patrons de procédé en tant que tels.

Enfin, la plupart des travaux existants sont de nature académique et ne visent pas une application (rapide) dans l'industrie. SPEM échappe bien sûr à cette critique puisque sa vocation est d'être une norme industrielle, mais la version 2 n'est pas encore officiellement adoptée. Si certains outils supportant SPEM commencent à apparaître sur le marché, ils sont loin de supporter toute la norme SPEM et notamment la partie concernant les patrons de procédé.

Notre proposition

Compte tenu des limites de l'état de l'art listées ci-dessus, l'objectif de notre travail est de mettre en œuvre efficacement la réutilisation de patrons de procédé dans la modélisation de procédés.

Pour cela, nous proposons de développer :

[1] Toutefois, le résultat d'une telle application dépend du choix de « bons » patrons par le concepteur. Le jugement relatif à la qualité du ou des patrons sélectionnés est une décision qui nécessite en partie l'intervention du concepteur, avec la dose inévitable de subjectivité associée.

- **un langage de description de procédés intégrant le concept de patron de procédé**

Un tel langage doit élargir la définition classique du concept de patron de procédé pour couvrir différents types de connaissances. Il doit fournir une description de patrons et de procédés suffisamment formelle pour que les procédés soient simulables/exécutables et les patrons applicables au moins de façon semi-automatique. Il doit enfin être fondé sur un LDP standardisé pour faciliter la diffusion et la communication des connaissances de procédé.

- **une méthode de modélisation de procédés pour guider systématiquement l'application des patrons de procédé**

Cette démarche doit couvrir les étapes de modélisation de procédés et être décrite sous forme d'un méta-procédé à grain fin pour donner aux concepteurs de procédé un guidage détaillé ainsi que pour faciliter sa mise en œuvre dans des outils support. Pour permettre une automatisation de l'application de patrons, ce méta-procédé devra s'appuyer sur un ensemble d'opérateurs exécutables de réutilisation/manipulation de modèles de procédé et de patrons de procédé.

La suite du document est dédiée à la présentation de l'approche adoptée pour réaliser cette proposition. Dans le chapitre 2, nous présentons la définition de notre LDP en utilisant la technique de la méta-modélisation. Dans le chapitre 3, nous décrivons notre méta-procédé et un ensemble d'opérateurs de réutilisation de patrons de procédé. Le chapitre 4 est consacré à la réalisation pratique de la proposition.

CHAPITRE II.

FORMALISATION DU CONCEPT DE PATRON DE PROCÉDÉ

L'état de l'art présenté dans le chapitre précédent a montré que l'approche à base de patrons est prometteuse pour la réutilisation de procédés. Pourtant, elle est encore appliquée d'une façon limitée et confuse à cause d'une formalisation insuffisante et en raison du manque de méthodes et d'outils support.

Pour pouvoir appliquer efficacement des patrons dans la modélisation de procédés, il faut d'abord avoir un formalisme qui permette de représenter des patrons de procédé ainsi que des modèles de procédé basés sur des patrons réutilisables. Il y a plusieurs travaux qui se concentrent sur ce besoin (c.f. Chapitre I). Cependant, ils se bornent souvent à formaliser des activités de développement réutilisables, et ne tiennent pas compte des autres types de connaissance plus abstraite sur la construction de procédé. En outre, les formalismes existants ne définissent pas suffisamment de moyens pour représenter efficacement la réutilisation (application) de patrons dans la modélisation de procédés. Les problèmes que nous venons d'évoquer nous ont conduit à la proposition d'une nouvelle approche de la notion de patron de procédé.

Dans ce chapitre, nous proposons une formalisation du concept de patron de procédé. Notre approche favorise la description et la réutilisation de plusieurs types de patrons de procédé à différents niveaux d'abstraction pour construire et pour améliorer des modèles de procédé.

Tout d'abord, nous expliquons le principe de l'approche (Section I), puis nous décrivons (section II) sa formalisation à travers un méta-modèle intégrant le concept de patron de procédé. En conclusion, nous résumons notre contribution vis-à-vis des travaux existants sur la formalisation de patrons de procédé.

I. NOTRE APPROCHE

Cette section vise à présenter, expliquer et discuter notre proposition de formalisation du concept de patron de procédé afin de permettre la modélisation de procédés fondée sur des patrons réutilisables.

Pour cela, nous précisons tout d'abord les définitions concernant la modélisation de procédés que nous adoptons dans cette thèse. Ensuite, nous présentons notre point de vue sur la définition, la classification et l'application de patrons de procédé. En dernier lieu, nous spécifions les caractéristiques attendues du formalisme proposé, et expliquons le choix de la technique employée pour sa mise en œuvre.

Dans un premier temps, les concepts introduits dans cette section et des exemples illustratifs sont représentés informellement. Une formalisation de ces concepts est ensuite présentée dans la section II.

I.1. NOTRE TERMINOLOGIE EN MODÉLISATION DE PROCÉDÉS

Malgré l'essor de la modélisation de procédés, il n'y a pas encore de cadre de référence standardisé pour les concepts du domaine. Dans l'état de l'art, nous avons présenté les définitions issues de la littérature. Nous précisons ici les définitions que nous adoptons pour les termes utilisés dans le reste de la thèse.

Procédé logiciel

> Un procédé logiciel définit les *tâches* à réaliser, les *rôles* participants et les *produits* manipulés pour élaborer ou maintenir un système logiciel.

Élément de procédé

> Les tâches, les rôles et les produits sont des éléments de procédé.

Processus logiciel

> Un processus logiciel est l'exécution d'un procédé logiciel pour réaliser un projet de développement logiciel.

Modèle de procédé[1]

> Un modèle de procédé est la représentation explicite d'un procédé dans un langage de description de procédés.

[1] Pour simplifier, par la suite, les procédés considérés sont implicitement des procédés logiciels.

Modélisation de procédé

> La modélisation d'un procédé consiste à définir un procédé et le représenter avec des modèles.

Méta-Modèle de procédé

> Un méta-modèle de procédé est un modèle définissant les concepts de base d'un langage de description de procédés.

I.2. CONCEPT DE PATRON DE PROCÉDÉ

Nous avons remarqué qu'il existe peu de définitions précises de la notion de patron de procédé. Généralement, les patrons de procédé sont considérés comme des patrons capturant des activités réutilisables de développement, et utilisés comme des modules de base pour construire de nouveaux procédés. Nous argumentons que cette définition n'est pas adéquate pour représenter l'idée originelle de patron de procédé qui vise à capturer et réutiliser diverses connaissances sur les procédés.

De telles connaissances peuvent aller d'expériences d'organisation et de gestion des processus de développement (par exemple, le choix d'un effectif approprié pour une organisation, le choix d'une durée convenable pour un projet [Coplien94]), à des solutions éprouvées de construction et de modélisation de procédés (par exemple, le procédé d'*inspection de code de Fagan* [Fagan86], des *modèles de Workflows*[v.d.Aalst03]). Les expériences d'organisation sont difficiles à formaliser et ne servent pas directement la modélisation de procédés. Nous les excluons donc des connaissances capturées par les patrons de procédé.

Pour nous, les patrons de procédé ont pour but de capitaliser et propager les savoir-faire de la modélisation de procédés. Plus précisément, les patrons de procédé capturent des connaissances qui peuvent être réutilisées pour élaborer des modèles de procédé d'une façon rapide et optimale. Nous arrivons donc à la définition suivante :

> Un patron de procédé se compose d'un problème de modélisation de procédés, d'une solution à ce problème et d'un contexte pour appliquer cette solution.

Nous caractérisons dans la suite les trois composants d'un patron de procédé : le problème, la solution, et le contexte.

I.2.1. Problème

> Le problème relatif à un patron de procédé traduit une question récurrente concernant la modélisation de procédés, afin de construire un nouveau procédé ou améliorer un procédé existant.

Pour caractériser les patrons de procédé et leur applicabilité dans la modélisation de procédés, nous identifions d'abord les problèmes habituels que les patrons de procédé cherchent à résoudre.

Après avoir étudié plusieurs questions qui se posent pendant la construction et la modélisation de procédés [Dewayne96][Scacchi99][Scacchi00][Rupprecht00][Madachy06], nous les classons en deux groupes : les questions concernant *la définition d'un procédé* et les questions concernant *l'organisation d'un procédé*. Nous définissons donc deux catégories de problèmes correspondant à ces deux groupes de questions :

Problème de définition d'un élément de procédé

Ce type de problème est représenté par la question *«Quel est le contenu d'un élément de procédé ?»*. Il s'agit donc du problème de définition soit d'une tâche, soit d'un rôle ou d'un produit. Ce type de problème se pose souvent dans la phase de conception de procédé. Il ne représente pas vraiment les difficultés, mais plutôt les questions ordinaires de la construction d'un procédé.

Par exemple, la tâche de révision d'un artéfact est quasiment indispensable dans le développement des logiciels. Donc la question *«Quelles sont les activités à faire lors de la tâche de révision technique d'un artéfact»* est forcément posée en définissant un procédé logiciel. Dans un autre contexte, si un concepteur veut construire un procédé basé sur le procédé RUP [Kruchten03], il peut poser des questions concernant la définition du rôle *RUP System Architect*, ou le contenu du produit *RUP Requirement Document*.

Problème d'organisation d'éléments de procédé

Nous classons dans cette catégorie les problèmes concernant la structuration d'un groupe d'éléments de procédé. Généralement, ce type de problème concerne la question *«Comment organiser un groupe d'éléments de procédé pour satisfaire un certain besoin ?»*. Cette question est posée dans le but de raffiner ou de modifier des éléments existants. La solution à un problème de ce type fournit un modèle pour enrichir ou (re)structurer des éléments du groupe.

Par exemple, un problème connu en développement de logiciels est celui posé par les informations qui apparaissent après que le processus ait déjà commencé et peuvent influencer des décisions antérieures dans le processus. La question *«Comment ordonner des tâches de développement pour prendre en compte des informations tardives ?»* est donc posée quand on veut construire un procédé efficace. La *modélisation des travaux coopératifs* est aussi un autre exemple de ce type de problème.

Les patrons de procédé peuvent être utilisés de différentes façons selon le problème traité. Les patrons traitant un problème de définition d'élément fournissent la connaissance nécessaire pour générer le contenu d'un élément de procédé. Par conséquent, ils peuvent servir de modules de base pour construire rapidement de nouveaux procédés. Quant aux patrons adressant un problème d'organisation d'éléments, ils fournissent un cadre général (template) pour structurer ou organiser des modèles de procédé d'une façon efficace.

I.2.2. Solution

La solution d'un patron fournit la connaissance nécessaire pour résoudre le problème de modélisation de procédés adressé par le patron. Si cette connaissance peut être exprimée avec des éléments de procédé et des relations entre eux, nous pouvons l'appliquer directement dans

la modélisation de procédés. Pour cette raison, nous représentons la solution d'un patron de procédé par un modèle de procédé.

> La solution d'un patron de procédé est un (fragment de) de modèle de procédé qui est destiné à être imité [1] pour contribuer à la modélisation d'un procédé.

Comme nous l'avons défini dans la section I.1, un modèle de procédé est la représentation explicite d'un procédé ; autrement dit, il représente des éléments de procédé et les relations entre eux. Un modèle peut représenter l'aspect structurel ou comportemental d'un (fragment de) procédé.

Modèle Structurel

Un modèle structurel reflète la décomposition d'un élément de procédé (par exemple *les sous-tâches d'une tâche*) ou des relations organisationnelles entre éléments de procédé (par exemple *des responsabilités de rôles sur des produits et des tâches*).

Modèle Comportemental

Un modèle comportemental montre l'aspect dynamique d'un élément de procédé (par exemple, *les états et transitions d'un produit*) ; ou des relations de coordination entre éléments de procédé (par exemple, *l'ordonnancement de tâches*).

I.2.3. Contexte

Chaque patron de procédé peut être appliqué dans un contexte donné.

> Le contexte d'un patron de procédé caractérise les conditions d'application du patron, les résultats attendus de cette application, et les situations appropriées pour l'application du patron.

En général, les conditions initiales et les conditions résultantes d'un patron portent sur des éléments de procédé (par exemple, l'état d'un produit, l'existence de rôles, l'accomplissement de tâches) ; les situations d'application d'un patron décrivent des caractéristiques de l'approche de développement, de l'organisation en équipe et du type de projet qui permet de tirer avantage de la solution du patron.

Par exemple, pour résoudre le problème de la modélisation des métiers d'une organisation (*business modeling* en anglais), on peut considérer deux patrons extraits de RUP [Kruchten03], *DomainModeling* et *BusinessModelling,* qui possèdent les contextes suivants :

[1] Certains auteurs utilisent le terme «instancier» en parlant de la réutilisation du modèle capturé dans la solution d'un patron. Pourtant, nous trouvons que ce terme ne reflète pas exactement la nature de cette action car le modèle réutilisé reste au même niveau d'abstraction que le modèle originel du patron. Nous adoptons donc le terme « imiter » proposé dans [Rieu99] : Imiter = Copier-Coller + Adapter.

	DomainModeling	***BusinessModeling***
Condition initiale	La spécification initiale de l'organisation est prête.	La spécification initiale de l'organisation est prête.
Condition résultante	Le *Domain Model* de l'organisation est élaboré.	Le *Business Vision Document*, le *Business Use-Case Model*, le *Business Analysis Model* de l'organisation sont élaborés.
Situation d'application	Avoir besoin d'information sur l'organisation concernant des entités métier, sans considérer le procédé du métier, pour développer des applications ayant comme but principal de présenter l'information.	Avoir besoin d'information sur le procédé métier de l'organisation pour l'établir ou l'améliorer en développant des applications supportant ce procédé métier.

I.3. TYPOLOGIE DES PATRONS DE PROCÉDÉ

Comme nous l'avons souligné dans l'état de l'art, les connaissances de procédé capturées dans les patrons peuvent être extraites de sources variées. Par exemple, si nous reconnaissons une structure similaire apparaissant dans plusieurs procédés sans considérer la différence sémantique, nous pouvons généraliser cette structure et la capturer dans un patron de procédé générique applicable à n'importe quel procédé (par exemple *la boucle interactive de tâches*). Nous pouvons également extraire des procédés d'une méthode de développement applicable pour divers projets et définir des patrons qui peuvent être appliqués à un certain type de procédé (par exemple, *la conception d'un système basé sur le cycle de vie cascade)*. Nous pouvons même définir un patron qui capture un procédé concret qui manipule un produit particulier (par exemple, *la révision technique d'un programme en Java*) applicable à des projets similaires.

Pour traduire la diversité des niveaux d'abstraction des connaissances représentées par les patrons de procédé, nous distinguons les trois types de patrons de procédé suivants :

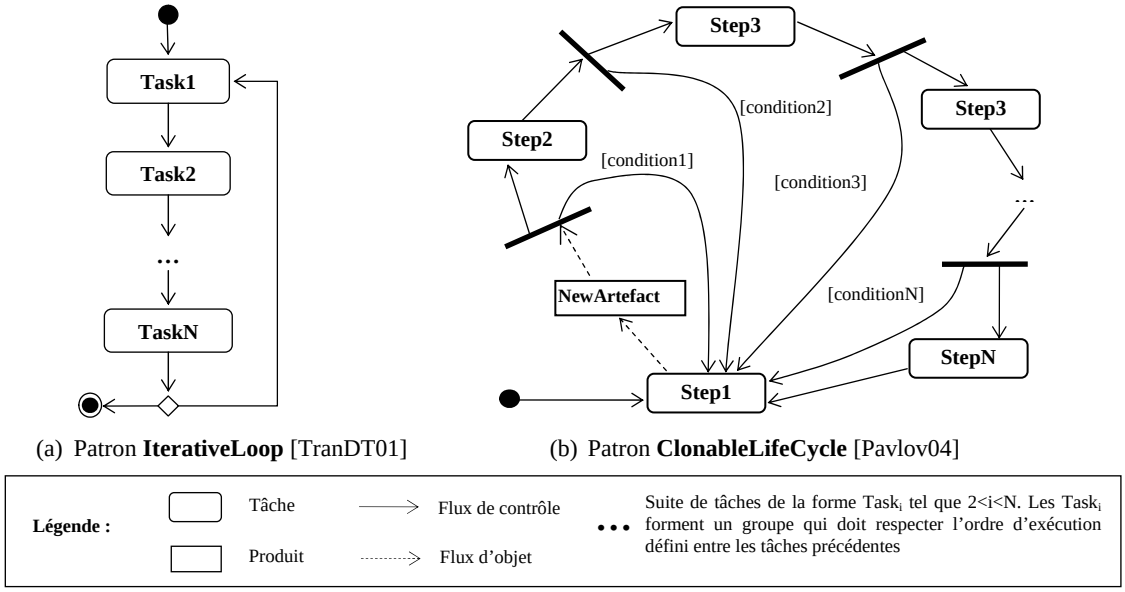
Patron Abstrait

> Un patron de procédé abstrait décrit une solution générique qui peut être (ré-)utilisée pour décrire une structure récurrente de procédé.

Les éléments d'un patron[1] abstrait sont simplement des représentants d'un type d'élément de procédé, ils n'ont pas de sémantique (par exemple, on ne connaît pas l'objectif d'une tâche *T*, ou l'usage d'un produit *P*). Dans un tel patron, ce n'est pas la sémantique précise des éléments qui compte, mais l'ensemble des éléments et les relations entre eux.

[1] Par simplification, dans la suite, le terme « élément d'un patron » est utilisé pour parler des éléments d'une solution d'un patron.

Les patrons abstraits peuvent être utilisés pour résoudre des problèmes d'organisation des éléments de procédé. Pour illustrer ce type de patrons de procédé, nous montrons deux exemples dans la Figure II-1.



(a) Patron **IterativeLoop** [TranDT01] (b) Patron **ClonableLifeCycle** [Pavlov04]

Figure II-1. Exemple de patrons abstraits

La Figure II-1a montre un patron abstrait capturant une structure générique pour décrire une boucle itérative de tâches. Cette structure générique est utilisable pour modéliser n'importe quel groupe de tâches exécutées de manière répétitive jusqu'à la vérification d'un critère ou l'obtention d'un résultat.

Le patron abstrait de la Figure II-1b propose une structure de procédé qui permet de retourner au début du processus pour traiter des informations tardives. Ce patron peut être appliqué quand on a besoin d'organiser des tâches d'un procédé pour qu'elles puissent prendre en compte de nouvelles informations qui arrivent pendant l'exécution du procédé.

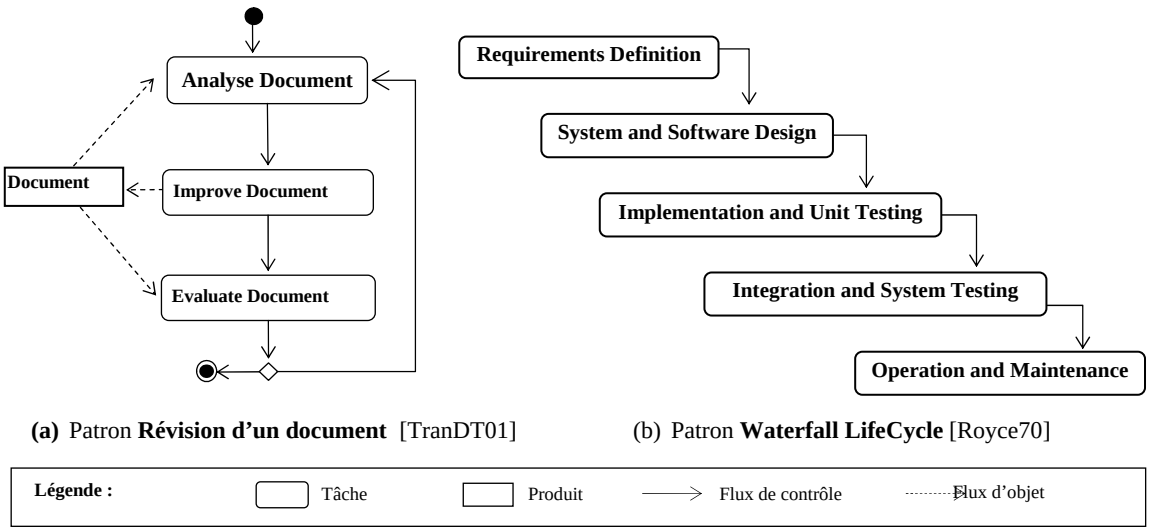
Parmi les patrons connus de la littérature, les patrons de workflow [v.d.Aalst03], les patrons modélisant des travaux collaboratifs [Lonchamp98], ou les patrons d'organisation de procédé métiers [Penker00], sont classés comme abstraits dans notre typologie.

Patron Général

> Un patron général décrit (une partie d') une méthode ou (d')une technique de développement destinée à être raffinée ultérieurement pour s'appliquer à différents projets.

Tout élément d'un patron général dispose d'une sémantique (c'est-à-dire qu'il a un nom, que l'on connaît son usage (par exemple la tâche *révision*, le produit *code*), mais il peut contenir des caractéristiques partiellement spécifiées qui devront être raffinées pour une utilisation particulière (par exemple, *réviser un certain artéfact ; un code dans un certain langage de programmation)*.

Un patron général peut s'appliquer pour définir le contenu d'un élément, ou pour organiser un groupe d'éléments. La Figure II-2 montre deux exemples de patrons généraux.



(a) Patron **Révision d'un document** [TranDT01] (b) Patron **Waterfall LifeCycle** [Royce70]

Figure II-2. Exemple de patrons généraux

Le patron de la Figure II-2a décrit une démarche générale pour réviser un document quelconque. La sémantique de ses éléments est définie. Cependant, pour pouvoir l'appliquer réellement en modélisant une tâche de révision dans un procédé concret, il faut spécifier le type de document à réviser pour pouvoir raffiner les actions de la tâche. Le patron de la Figure II-2a est donc un patron général.

La Figure II-2b montre un patron général capturant un cycle de vie classique qu'on peut appliquer pour organiser les phases de développement d'un logiciel. Cependant, le patron ne décrit pas les détails de chaque phase, pour qu'on puisse le raffiner et l'utiliser dans plusieurs projets de développement pour fabriquer divers logiciels.

La plupart des patrons représentant des méthodes de développement comme OSSP [Ambler98], OPF [OPF] ou UP [Jacobson99] sont des patrons généraux.

Patron Concret

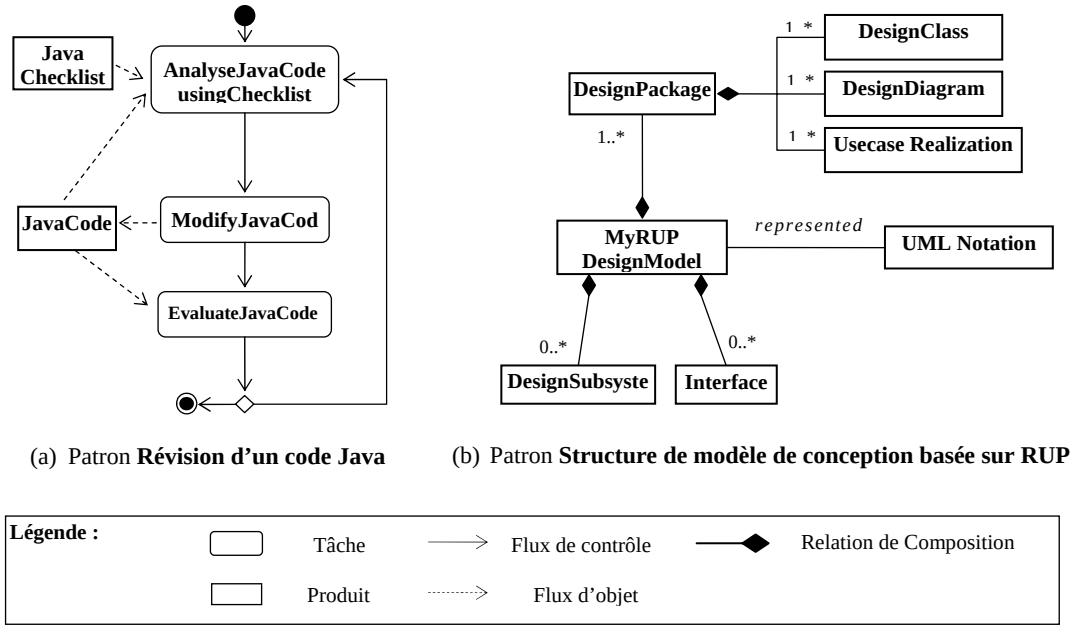
> Un patron concret capture une solution de procédé bien spécifique qui peut être appliquée à un type de projet particulier.

Un patron concret contient des éléments complètement définis, c'est-à-dire qu'ils possèdent une sémantique précise, et sont représentés avec des formalismes concrets[1].

Un patron concret peut être utilisé pour définir le contenu d'un élément, ou pour organiser un groupe d'éléments.

[1] Un formalisme concret est une convention de représentation utilisée pour décrire un artéfact. Ce peut être un langage de description (par exemple UML), un langage de programmation (par exemple Java) ou juste un format de représentation (par exemple un template de représentation de spécification des besoins).

Nous montrons dans la Figure II-3 deux exemples de patron concret.



(a) Patron **Révision d'un code Java** (b) Patron **Structure de modèle de conception basée sur RUP**

Figure II-3. Exemple de patrons concrets

La Figure II-3a montre un procédé concret utilisé pour vérifier un code Java en employant une *Checklist* pour analyser le code. Les tâches de ce procédé sont spécifiées par des actions concrètes[1]. Ce patron est prêt à être appliqué pour des projets de développement utilisant Java comme langage de codage.

La Figure II-3b décrit la structure d'un modèle de conception spécialement défini par une équipe de développement. Cette équipe a adapté la structure de modèle de conception du procédé RUP [Kruchten03] à ses besoins, en utilisant la notation UML et les templates de document définis par le RUP. Ce patron reflète donc la solution particulière de cette équipe.

Pour supporter la description des différents types de patrons introduits ci-dessus, il faut fournir des moyens pour représenter des modèles de procédé à différents niveaux d'abstraction. Pour cela, nous distinguons également trois niveaux d'abstraction pour les éléments de procédé : *abstrait*, *général* et *concret*. Cette hiérarchisation sera appliquée à tous les types d'éléments, c'est-à-dire aux *produits*, *rôles* et *tâches*.

Élément de procédé Abstrait

> Un élément de procédé abstrait n'a aucune définition précise.

Par exemple, on peut considérer une tâche *T*, un produit *P* ou un rôle *R* sans préciser leur sémantique.

Les éléments abstraits représentent des éléments dans un procédé quelconque. Nous les utilisons pour décrire des solutions de procédé génériques.

[1] Pour simplifier l'exemple, on ne montre pas ces actions concrètes ici.

Élément de procédé Général

> Un élément de procédé général est partiellement défini avec une sémantique qui peut être encore raffinée.

Par exemple, si on parle d'un *code*, on sait que c'est un artéfact de développement qui contient des instructions générant un programme logiciel ; pourtant, on ne sait pas encore dans quel langage ce code est décrit. Un code, dans ce cas, est un *produit général*. Quand on considère une *tâche de révision*, on sait que son but est de vérifier la correction d'un artéfact, mais on ne peut pas décrire comment elle fonctionne, car le produit manipulé par cette tâche n'est pas encore spécifié. Cette tâche est donc une tâche générale qui pourra être spécialisée plus tard en *réviser un code* ou *réviser des exigences*, etc.

Les éléments généraux représentent des éléments d'une méthode de développement qui pourra s'adapter à un but spécifique. Nous les utilisons dans la description des procédés applicables à plusieurs projets fabriquant divers produits.

Élément de procédé Concret

> Un élément de procédé concret est totalement défini.

Par exemple, un *codeJava* est un *produit concret* parce qu'on sait que c'est un programme objet codé dans un langage doté d'une grammaire précise. Par conséquent, on peut définir les détails de la *tâche de révision d'un code Java* pour la rendre également concrète.

Les éléments de procédé concrets sont utilisés pour décrire une approche de développement spécifique pour fabriquer des produits spécifiques.

En raffinant[1] un élément abstrait, on obtient un élément général ou concret. En raffinant un élément général on peut obtenir un élément général ou un élément concret. Un élément concret ne peut plus être raffiné.

La Figure II-4 montre la hiérarchie des niveaux d'abstraction des éléments de procédé dans notre approche et les transitions possibles entre les différents niveaux.

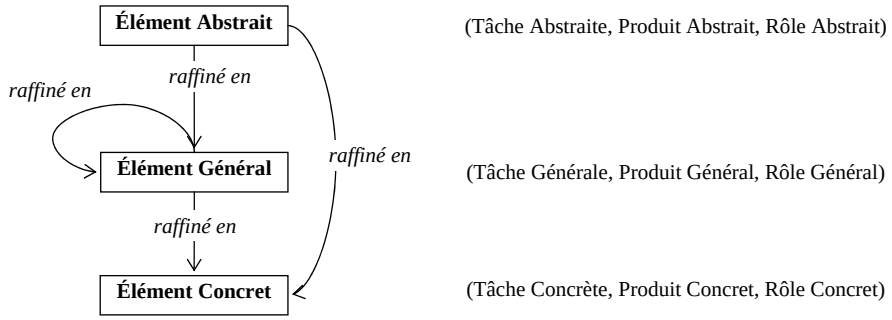


Figure II-4. Relations entre niveaux d'abstraction des éléments de procédé

[1] Nous utilisons le terme «raffiner un élément» dans le sens d'enrichir la description de l'élément pour le rendre plus spécifique. Cependant, un raffinage ne change pas la nature de l'élément. Par exemple, en raffinant une tâche, on obtient une autre tâche plus spécifique, mais pas un produit ou un rôle.

Nous expliquons en détail l'application de cette hiérarchisation à chaque type d'élément de procédé dans la section II.1 ci-dessous.

Un patron[1] abstrait doit contenir au moins un élément abstrait. Un patron général n'a aucun élément abstrait, et contient au moins un élément général. Un patron concret ne contient que des éléments concrets. La Figure II-5 montre les compositions possibles d'éléments pour former des patrons à différents niveaux d'abstraction.

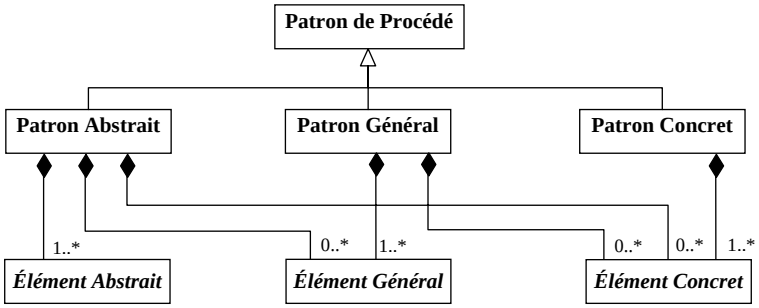


Figure II-5. Éléments constitutifs des patrons aux différents niveaux d'abstraction

La classification de patrons de procédé selon leur niveau d'abstraction est une originalité de notre approche. Comme la plupart des travaux sur les patrons de procédé, notre approche supporte les patrons capturant des procédés réutilisables, qui se situent aux niveaux général et concret. Par contre, avec les patrons abstraits, nous permettons également de représenter et réutiliser des connaissances de procédé conceptuelles, connaissances qui ne sont pas formalisées dans les travaux existants[2] (c.f. Chap 1).

I.4. APPLICATIONS DE PATRONS DE PROCÉDÉ

Pendant la modélisation de procédés, plusieurs scénarios s'offrent aux concepteurs pour réutiliser des patrons de procédé. Nous avons identifié deux utilisations principales des patrons de procédé qui répondent respectivement aux deux types de problèmes discutés dans la section I.2.1 : la définition d'un élément de procédé, et l'organisation d'éléments de procédé.

Aide à la définition d'un élément de procédé

> Un patron de procédé peut être employé en tant que *template* pour produire le contenu d'un élément de procédé.

Plus précisément, si un patron adresse la question «Quelle est la définition de l'élément *X* ?», on peut imiter le modèle de procédé capturé dans ce patron pour générer la description de l'élément *X* (ou des éléments raffinés de *X*).

Dans la littérature, on peut trouver la relation *binding* qui exprime le lien de traçabilité entre un modèle et le template utilisé pour sa conception [UML05b]. Nous adaptons cette

[1] Pour simplifier, nous utilisons le terme de « patron » à la place de celui de « solution de patron ».

[2] Comme nous l'avons cité précédemment, il y a des travaux qui identifient des patrons que l'on peut qualifier d'abstraits [Penker00][v.d.Aalst03][Lonchamp98], mais ils ne les formalisent pas dans un objectif de réutilisation.

relation générale à l'application de patrons de procédé pour générer le contenu d'un élément de procédé. Sur un modèle de procédé, on pourra représenter cette application par une relation *binding* reliant un élément de procédé à un patron. Nous formaliserons cette relation *binding* dans la section II.3.1.2. Dans le même but, PROMENADE propose la relation *PPBinding* représentant le lien entre un modèle de procédé et un patron de procédé qu'il instancie. Cependant, la sémantique précise de cette instanciation ainsi que l'opérateur réalisant un telle instanciation n'a pas été définie.

Pour illustrer cette utilisation de patrons, nous présentons dans la Figure II-6 un exemple simplifié de la modélisation d'un procédé de développement d'une compagnie. Dans la Figure II-6a un procédé général contient des éléments dont les contenus ne sont pas encore spécifiés en détail. Pour raffiner ce procédé, le concepteur peut réutiliser des patrons appropriés pour élaborer les descriptions des tâches. Dans cet exemple, on suppose que le concepteur a trouvé le patron *TechnicalReview* (Figure II-6b) qui capture un modèle décrivant la révision d'un artéfact. Il peut appliquer ce patron pour générer les contenus de deux tâches de révision du procédé : *DesignReview* et *CodeReview*. Le résultat de ces applications est montré dans la Figure II-7

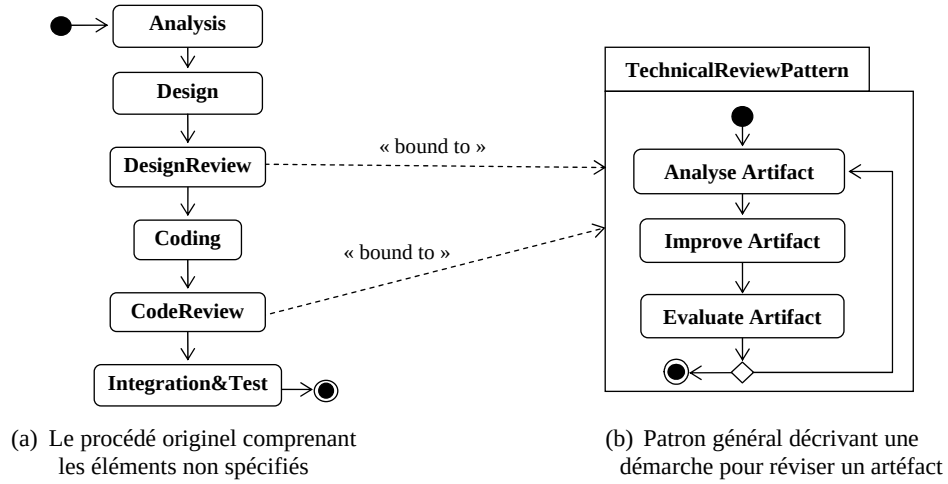


Figure II-6. Exemple d'application d'un patron pour définir un élément de procédé

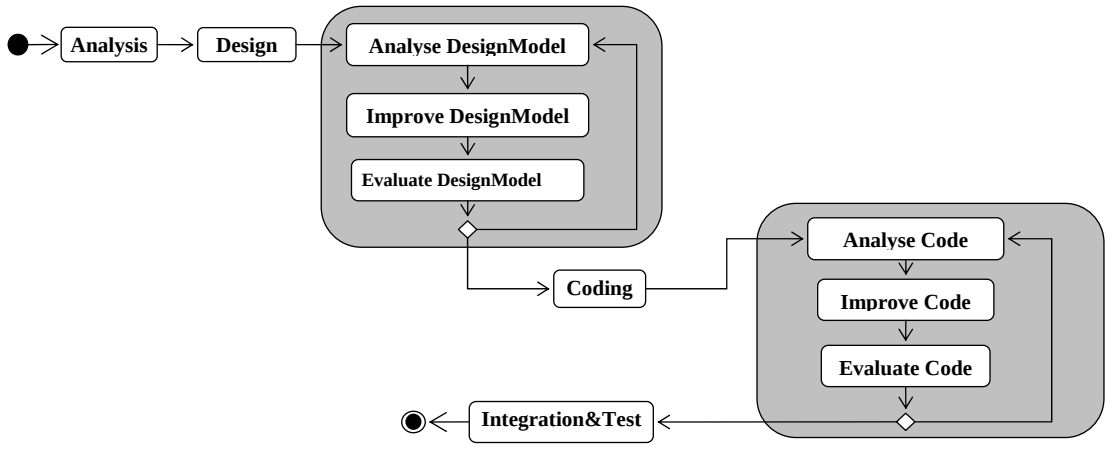


Figure II-7. Résultat de l'application du patron *TechnicalReview* définissant les tâches *DesignReview* et *CodeReview*

L'application des patrons pour définir des éléments de procédé est l'utilisation la plus classique, et est, à notre connaissance, l'unique moyen employé par la communauté informaticienne pour appliquer des patrons de procédé dans la modélisation de procédés[1].

Pourtant, pendant la modélisation de procédés, surtout dans la phase d'amélioration de modèles de procédé, les concepteurs doivent parfois restructurer ou enrichir un fragment de modèle pour satisfaire certaines contraintes ou pour accroître l'efficacité du procédé. Plus précisément, ils peuvent vouloir appliquer de nouvelles relations sur un groupe d'éléments existants, ou ajouter de nouvelles informations à ce groupe. Dans de tels cas, une autre manière d'appliquer les patrons de procédé est nécessaire. Il s'agit de l'utilisation des patrons pour (ré)organiser ou pour enrichir un groupe d'éléments de procédé. Ce genre d'utilisation de patrons de procédé n'a pas été encore proposé dans la littérature du domaine. Cette idée est abordée seulement dans [Iida99] mais l'auteur n'a pas défini de mécanisme pour supporter cette utilisation.

Pour rendre applicable cette utilisation de patrons de procédé, nous la définissons comme ci-dessous :

Aide à l'organisation ou à l'enrichissement d'un groupe d'éléments de procédé

> Un patron de procédé peut être employé pour enrichir ou pour (ré)organiser un groupe d'éléments de procédé existants.

Plus précisément, si un patron capturant un modèle S contient des éléments $\{s_1,s_2,..s_N\}$, on peut appliquer le modèle S à un groupe d'éléments existant $\{d_1,d_2,..d_M\}$ pour ajouter une nouvelle couche d'information à ce dernier. Dans une telle application, chaque d_i joue le rôle d'un s_i dans le patron. Par conséquent, les caractéristiques (attributs, relations) définies sur s_i se transmettront à d_i. La condition générale de cette application est que d_i doit être conforme[2] à s_i.

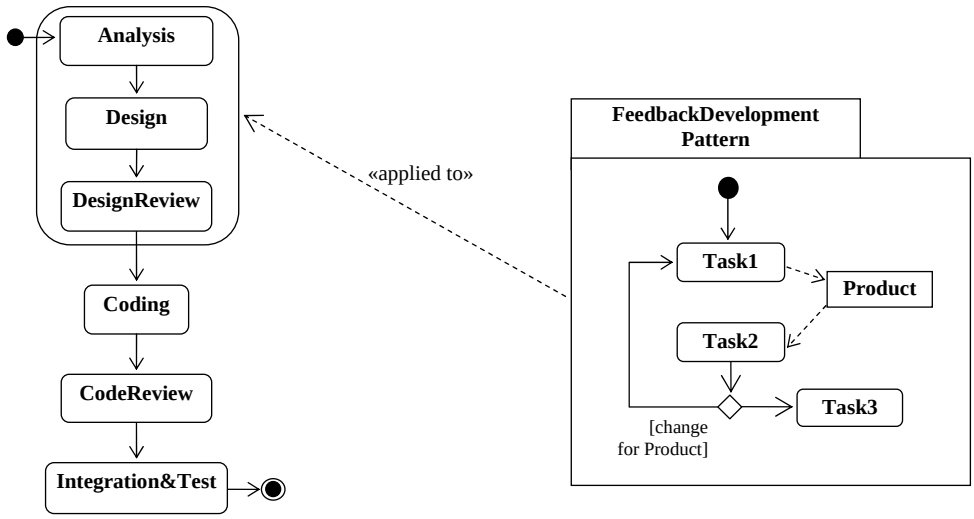
Nous proposons une nouvelle relation, *applying*, pour représenter l'application d'un patron de procédé à un groupe d'éléments de procédé dans le but de l'enrichir ou de le (ré)organiser. La relation *applying* exprime le lien entre le modèle source du patron et les éléments cibles à (re)structurer d'après le modèle source. Nous formaliserons cette relation *applying* dans la section II.3.1.3.

La Figure II-8 montre un exemple d'application de patron pour restructurer des éléments de procédé. Nous reprenons le procédé de la Figure II-6a pour illustrer cette utilisation de patron. Supposons que tous les éléments du modèle de la Figure II-8a soient raffinés. Le concepteur veut améliorer ce modèle de procédé parce que son cycle de vie actuel ne permet pas de prendre en compte des changements durant le développement. Par exemple, en exécutant le procédé représenté par le modèle de la Figure II-8a, si des modifications de besoins surviennent

[1] Plus précisément, la communauté parle simplement de l'utilisation de tâches réutilisables comme des blocs de base pour construire des procédés. Nous considérons cela comme l'application de patrons pour définir des tâches. Par contre, nous élargissons cette application pour pouvoir définir également des rôles et des produits à partir des patrons.

[2] En considérant le type et le niveau d'abstraction.

pendant la tâche *Design*, les développeurs ne peuvent pas revenir à la tâche *Analysis* pour traiter ces nouveaux besoins car le modèle de procédé n'autorise pas un tel retour.



(a) Procédé originel ayant le cycle de vie séquence

(b) Patron décrivant une structure de procédé permettant un ordre d'exécution avec des rétroactions entre deux tâches

Figure II-8. Exemple d'application d'un patron pour organiser des éléments

Dans cette situation, le concepteur peut réorganiser les éléments existants du procédé originel par emploi du patron *FeedbackDevelopment* qui capture une structure de procédé générique représentant un ordre d'exécution avec des rétroactions entre deux tâches (Figure II-8b). L'exemple montre une telle application pour restructurer le groupe de trois tâches *Analysis*, *Design* et *DesignReview*. La Figure II-9 montre le procédé résultant de cette application. Elle met en évidence l'ajout du retour entre les tâches *Design* et *Analysis*.

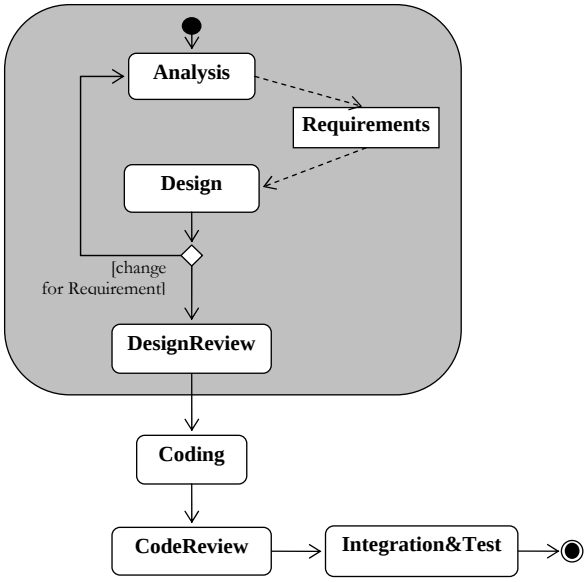


Figure II-9. Résultat de l'application du patron *FeedbackDevelopment* sur le modèle de procédé de la Figure II-8a

Dans notre hiérarchie de patrons de procédé (c.f.Figure II-4), plus haut est le niveau où se situe un patron, plus il est réutilisable. C'est-à-dire qu'un patron abstrait peut être appliqué pour définir ou réorganiser des éléments généraux ou concrets ; à son tour, un patron général est applicable aussi pour des éléments généraux ou concrets ; mais un patron concret ne peut s'appliquer qu'à des éléments concrets. Pour permettre une telle utilisation, nous employons le mécanisme de paramétrisation pour formaliser le concept de patrons de procédé. Nous présenterons cette formalisation dans la section II.3.1.1.

I.5. PRINCIPE DE FORMALISATION DU CONCEPT DE PATRON DE PROCÉDÉ

Dans la section précédente, nous avons présenté informellement nos définitions concernant le concept de patron de procédé. Nous expliquons dans cette section l'approche utilisée pour formaliser ces définitions. Nous précisons d'abord les objectifs de la formalisation, puis nous justifions le choix de la technique de formalisation.

Objectifs de la formalisation

Le but de la formalisation du concept de patron de procédé est de permettre une utilisation explicite des patrons dans la modélisation de procédés.

Plus précisément, nous voulons *définir un langage de description de procédés intégrant le concept de patron de procédé*. De plus, pour faciliter la compréhension, l'échange d'information ainsi que le développement d'outils support, le langage proposé devra être *conforme aux standards du domaine*, et *rigoureusement défini*.

Choix de la technique de formalisation

Pour atteindre les objectifs ci-dessus, nous avons fait les deux choix suivants :

1er choix : S'inspirer de SPEM 1.1 [SPEM05] pour la description de procédés

Dans l'état de l'art, nous avons soulevé le problème de la diversité des langages de modélisation de procédés. Il y a plusieurs formalismes proposés, chacun définissant des notions différentes, bien que souvent très proches. Cela empêche le partage et l'échange d'information entre différents travaux. Pour éviter ce problème, nous voulons fonder notre proposition sur un formalisme de description de procédés largement accepté par la communauté.

Initialement, nous avons eu l'intention d'intégrer notre définition de patrons de procédé à SPEM, un standard de l'OMG dédié à la description de procédés logiciels. Mais, la version courante 1.1 de SPEM comporte certaines lacunes [SPEM04] et est en train d'évoluer vers une nouvelle version 2.0 [SPEM07]. D'ailleurs, SPEM ne fournit pas assez de moyens pour exprimer notre proposition, car il ne tient pas compte de la représentation de modèles à plusieurs niveaux d'abstraction.

Nous avons donc décidé de nous inspirer du modèle conceptuel de SPEM 1.1 [SPEM05] concernant la définition des éléments de procédé.

2e choix : Formaliser la description de procédés et de patrons de procédé par un méta-modèle de procédé conforme à MOF 2.0 [MOF06]

Dans le contexte évolutif de l'ingénierie dirigée par les modèles, il nous apparaît évident que les procédés doivent être décrits sous forme de modèles. Cela nous permet de profiter des principes généraux de l'ingénierie des modèles, c'est-à-dire la représentation de procédés à différents niveaux d'abstraction, la possibilité de vérification et de transformation des modèles de procédé, etc.

Nous pensons donc que la méta-modélisation est un moyen pertinent pour formaliser les concepts de procédé et de patron de procédé. Cette technique, plus particulièrement l'approche MDA de l'OMG [MDA01], permet de définir un langage par un méta-modèle explicite basé sur le noyau commun MOF 2.0 *(Meta Object Facility)*[MOF06] et représenté par le langage universel UML [UML05a]. Définir notre formalisme comme un méta-modèle dans le cadre de l'architecture MDA facilite sa compréhension, sa diffusion, son utilisation et son alignement avec des travaux relatifs grâce à la notation standardisée et la disponibilité d'outils support.

Il y a deux façons de définir un méta-modèle basé sur UML 2.0 : la création d'un profil ou le développement d'un méta-modèle indépendant.

L'intérêt de l'approche par profil est de pouvoir profiter de la riche capacité d'expression de UML2.0 SuperStructure [UML05b] ainsi que des outils de modélisation supportant ce standard. L'inconvénient de cette approche est de devoir définir de nombreuses contraintes pour préciser la sémantique et l'application des concepts génériques de UML2.0 réutilisés dans le profil. Par conséquent, la spécification d'un formalisme sous forme d'un profil devient plus difficile à comprendre, car il demande la connaissance du méta-modèle de base (ici UML2.0) et la définition des contraintes à ajouter.

La définition quant à elle d'un méta-modèle conforme à MOF [MOF06] par extension de UML2.0 InfraStructure[1] [UML05a], permet d'avoir à la fois la standardisation et la liberté de définir des concepts et des représentations spécialisées du domaine.

Dans le cadre de cette thèse, nous avons voulu décrire notre formalisme de la façon la plus simple et la plus autonome possible pour focaliser notre attention sur les concepts spécialisés de procédé et de patron de procédé. En conséquence, nous avons décidé de développer un méta-modèle indépendant nommé **UML-PP** (*UML for Process Pattern*).

La Figure II-10 montre le positionnement de notre proposition dans l'architecture MDA et les relations de notre méta-modèle avec les standards concernés.

[1] Afin de faciliter la réutilisabilité, UML 2.0 propose le noyau commun *Core* (c.f. paquetage UML2.0 InfraStructureLibrary [UML05a]) conçu pour permettre de développer de nouveaux langages basés sur le même pivot que UML. *Core* est un méta-modèle définissant un ensemble de concepts de base de la modélisation. En réutilisant l'ensemble ou une partie de *Core* pour construire un nouveau méta-modèle, on peut bénéficier de la syntaxe abstraite et de la sémantique déjà définies.

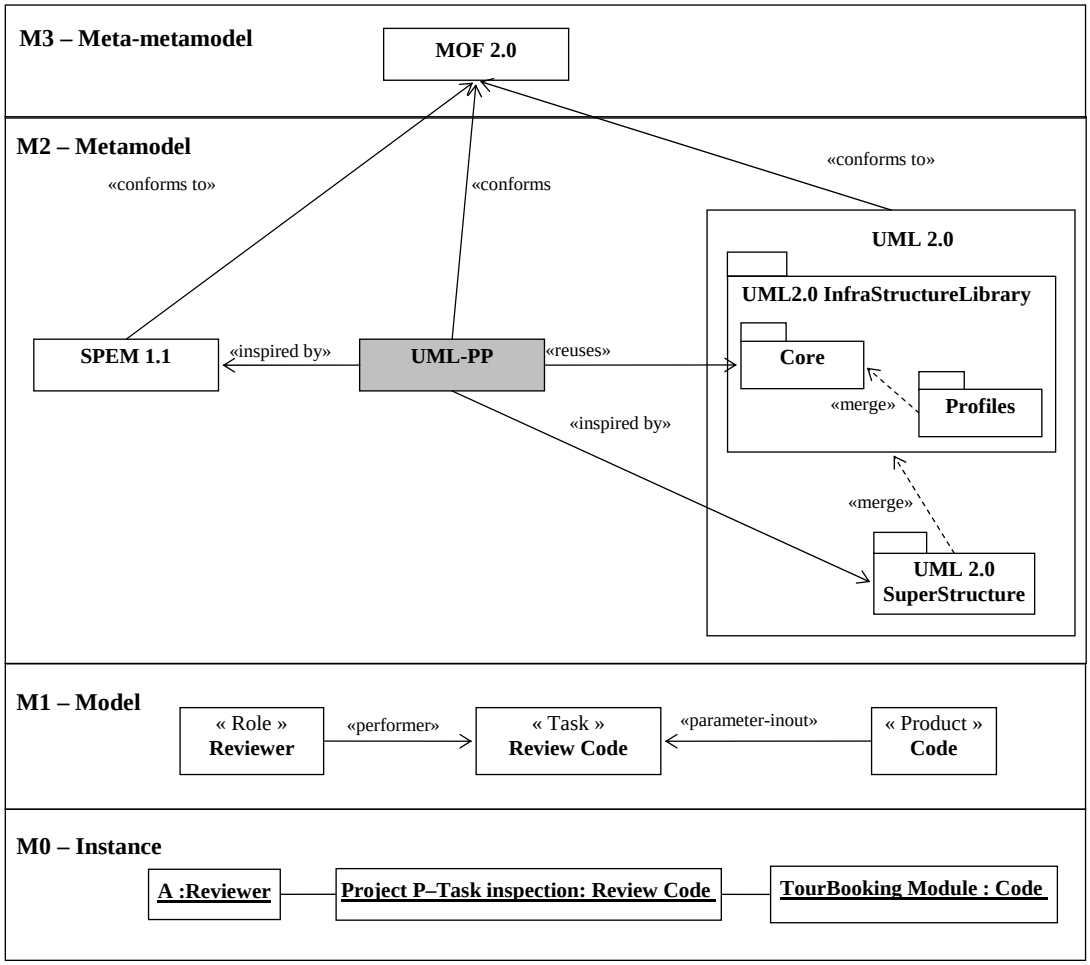


Figure II-10. Positionnement du méta-modèle UML-PP au sein de l'architecture MDA

Dans l'architecture à quatre couches du MDA, le niveau le plus élevé (M3) correspond à l'unique méta-méta-modèle MOF ; le niveau en dessous (M2) est celui des différents méta-modèles, y compris UML2.0, définissant chacun un formalisme propre à un domaine particulier ; au niveau inférieur (M1) se trouvent les modèles, chaque modèle étant basé sur un seul méta-modèle ; au niveau le plus bas (M0) se trouvent les instances des modèles.

Notre méta-modèle UML-PP se situe au niveau M2, le même niveau que UML2.0 et SPEM 1.1, et réutilise le paquetage *Core* de UML2.0 InfraStructure. De plus, UML-PP s'inspire du modèle conceptuel de SPEM1.1 et de la syntaxe abstraite de UML2.0 SuperStructure. Le méta-modèle UML-PP définit les concepts de base pour décrire les procédés et les patrons de procédé, par exemple *Task*, *Rôle*, *Product* et *ProcessPattern*.

Un modèle de procédé conforme au méta-modèle UML-PP est classé au niveau M1. Les éléments d'un tel modèle sont des instances des méta-classes définies par UML-PP.

Finalement, une instance d'un modèle de procédé pour un projet concret se trouve au niveau M0. Elle contient des objets instanciés des classes du modèle de procédé du niveau M1.

II. LE MÉTA-MODÈLE UML-PP *(UML FOR PROCESS PATTERNS)*

Nous avons présenté notre approche de formalisation du concept de patron procédé dans la section précédente. Nous décrivons dans cette section le méta-modèle UML-PP *(UML for Process Patterns)* que nous avons développé pour permettre la description de procédés fondée sur les patrons de procédé.

Le méta-modèle UML-PP permet d'une part de décrire les éléments principaux des procédés logiciels, et d'autre part de représenter les procédés basés sur les patrons de procédé.

Nous organisons notre méta-modèle en trois paquetages : le paquetage *ProcessStructure* décrit les éléments constituant un procédé ; le paquetage *ProcessPattern* décrit la structure d'un patron de procédé ; le paquetage *PatternRelationship* décrit les mécanismes d'application de patrons ainsi que les relations entre les patrons. Ces paquetages sont reliés par les dépendances *« import »*[1] comme montré dans la Figure II-11 :

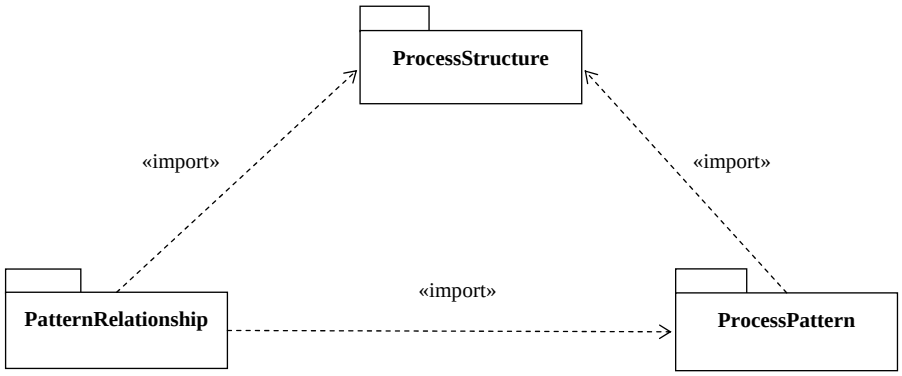


Figure II-11. Organisation des paquetages du méta-modèle UML-PP

Ces paquetages sont décrits en détail dans la suite de cette section. Pour chaque paquetage, nous décrivons d'abord sa syntaxe abstraite, ensuite sa sémantique, et en dernier lieu, sa syntaxe concrète.

La syntaxe abstraite est représentée sous forme de diagrammes de classes UML. Dans ces diagrammes, les éléments en gris correspondent aux éléments issus du méta-modèle UML2.0, les autres correspondent aux concepts que nous avons ajoutés afin de permettre l'expression des éléments de procédé et des patrons de procédé.

La sémantique de chaque élément du méta-modèle est décrite d'abord en langage naturel, renforcé par des règles de bonne modélisation (*Well-formedness rules*) exprimées en langage OCL [OCL05]. Dans ce chapitre, nous donnons seulement la sémantique statique du méta-modèle qui explique la signification de chaque élément et fournit les règles pour vérifier la cohérence des modèles bâtis avec les concepts UML-PP. La sémantique dynamique concernant l'application des patrons de procédé sera présentée dans le chapitre III.

[1] L'importation de paquetage est une relation orientée qui permet à un paquetage d'ajouter le contenu d'un autre paquetage à son espace de nommage.

Pour être compatible au maximum avec UML, nous définissons la syntaxe concrète d'UML-PP en utilisant les stéréotypes d'UML. Autrement dit, la notation représentant des instances d'une méta-classe UML-PP est adaptée de celle de sa classe de base UML et précisée avec un mot-clé (déduit du nom de la méta-classe UML-PP). Pour certains concepts, nous proposons également des options alternatives dans les cas où il n'est pas nécessaire de montrer les détails des éléments modélisés.

II.1. PAQUETAGE PROCESSSTRUCTURE

Ce paquetage contient les concepts[1] qui définissent les éléments de procédé (*ProcessElement*) et les relations entre eux (*ProcessRelation*). Ces concepts ont été extraits de plusieurs travaux réalisés dans ce domaine, et inspirés par la norme SPEM 1.1[SPEM05].

Nous décrivons un procédé à l'aide de trois types d'éléments principaux : les *tâches*[2] (*Task*) à réaliser, les *rôles*[3] (*Role*) participants et les *produits (Product)*[4] manipulés.

La Figure II-12 montre ces éléments et les relations entre eux.

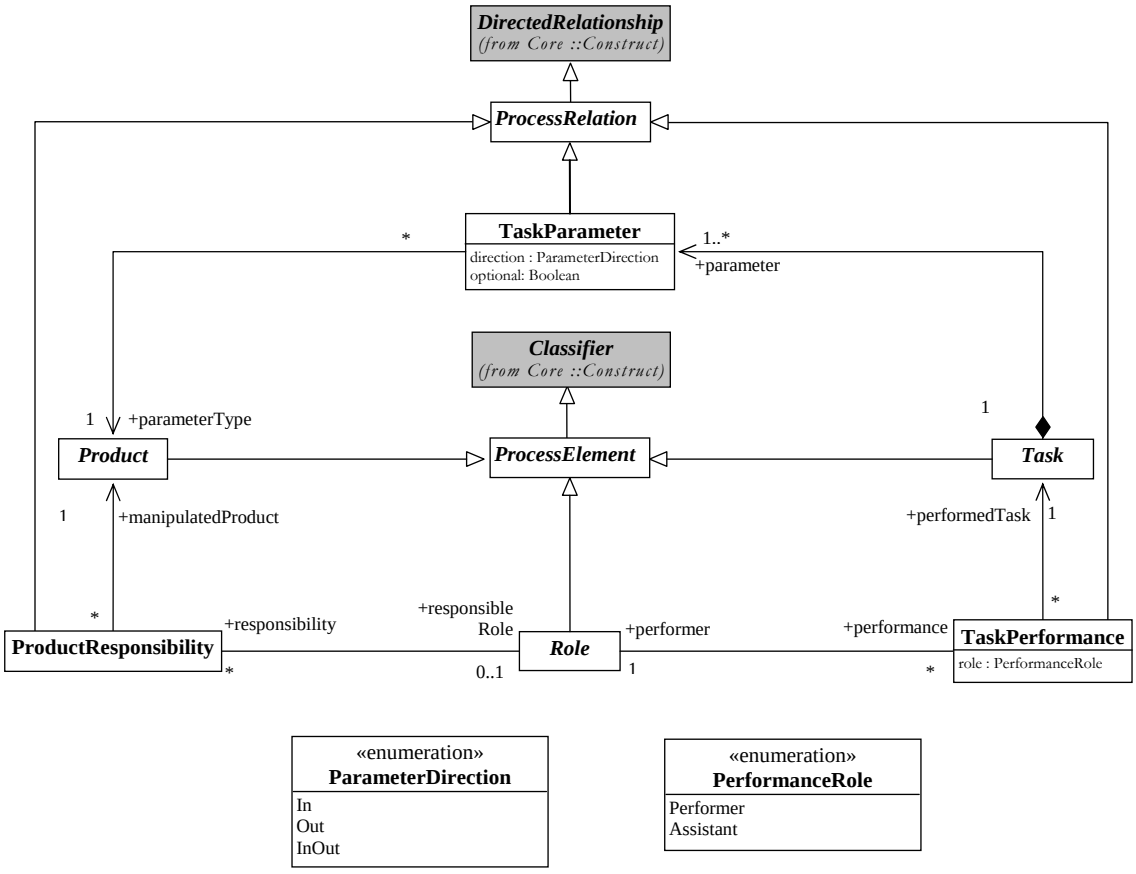


Figure II-12. Méta-modèle d'éléments de procédé

[1] Dans la description du méta-modèle UML-PP, nous utilisons le terme de «concept» dans le sens «élément de modélisation»

[2] Ce concept remplace les concepts de *WorkDefinition* et *Activity* de SPEM

[3] Ce concept remplace les concepts de *ProcessPerformer et ProcessRole* de SPEM

[4] Ce concept remplace le concept de *WorkProduct* de SPEM

II.1.1. Élément de Procédé

Nous définissons le concept abstrait *ProcessElement* pour représenter une généralisation des éléments qui font partie des procédés. Les concepts de produit, de rôle et de tâche sont décrits par des sous-classes de *ProcessElement* (Figure II-12).

Dans la section I.3 nous avons présenté l'intérêt de la description des éléments à différents niveaux d'abstraction, et proposé une hiérarchie à trois niveaux : *abstrait*, *général* ou *concret*. Nous montrons dans la Figure II-13 l'extrait du méta-modèle définissant les éléments à ces niveaux d'abstraction.

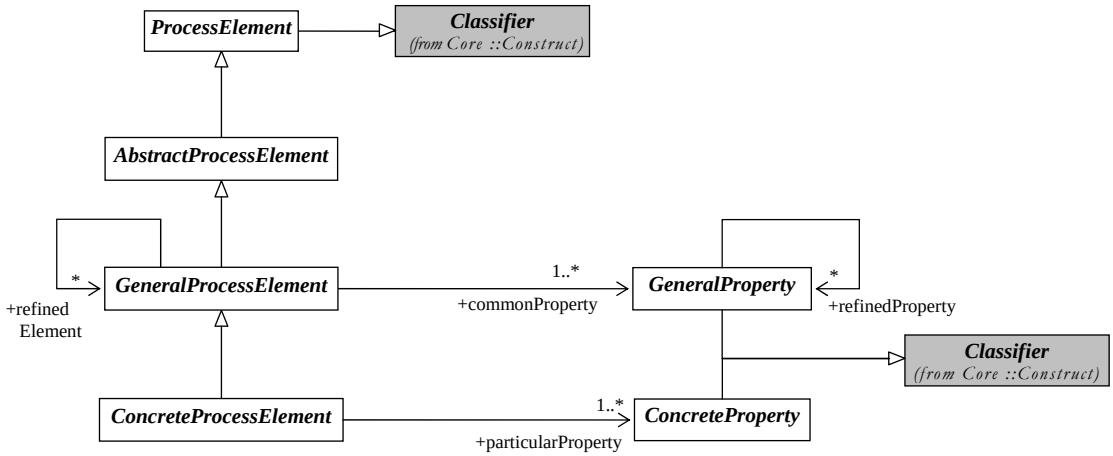


Figure II-13. Éléments de procédé à différents niveaux d'abstraction

AbstractProcessElement

Un élément abstrait est une généralisation des éléments de procédé qui sont classés dans un type d'élément (c.-à-d. *tâche*, *produit* ou *rôle*), mais n'a pas d'autres caractéristiques. Il ne possède donc aucune sémantique.

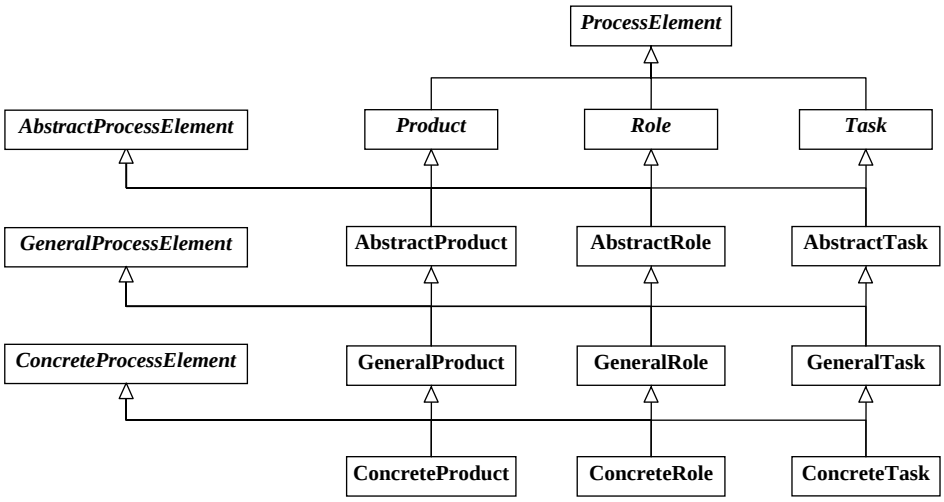
GeneralProcessElement

Un élément de procédé général est caractérisé par un type d'élément de procédé, et est décrit en plus par des caractéristiques (*GeneralProperty*) qui lui donnent une sémantique générale. Ces caractéristiques peuvent être raffinées pour décrire un autre élément plus spécifique.

ConcreteProcessElement

Un élément de procédé concret est complètement défini ; il est classé dans un type d'élément de procédé, et a une sémantique spécifiée par des caractéristiques concrètes qui ne peuvent plus être raffinées.

La Figure II-14 montre l'application de cette hiérarchie sur les concepts de produit, de rôle et de tâche.



.

Figure II-14. Hiérarchie d'éléments de procédé

Opérations additionnelles

Afin de faciliter la vérification de la cohérence des éléments, nous définissons les méta-opérations[1] suivantes sur un élément de procédé :

[OP1] La méta-opération *getAbstractionLevel()* renvoie le niveau d'abstraction d'un élément de procédé.

```
ProcessElement::getAbstractionLevel() : String;
if (self.oclIsTypeOf(AbstractProduct)or self.oclIsTypeOf(AbstractRole)
   or self.oclIsTypeOf(AbstractTask))
   getAbstractionLevel = 'Abstract'
else if (self.oclIsTypeOf(GeneralProduct)or
     self.oclIsTypeOf(GeneralRole)or self.oclIsTypeOf(AbstractTask))
     getAbstractionLevel = 'General'
     else if (self.oclIsTypeOf(GeneralProduct)or
         self.oclIsTypeOf(GeneralRole)or self.oclIsTypeOf(AbstractTask))
         getAbstractionLevel = 'Concrete'
        endif
   endif
endif
```

[OP2] La méta-opération *isIdenticalTo()* détermine si un élément a le même type qu'un autre.

```
ProcessElement::isIdenticalTo(p : ProcessElement) : Boolean;
isIndenticalTo = p->oclIsTypeOf(self.oclType)
```

[OP3] La méta-opération *isCompatibleWith()* détermine si un élément est compatible avec un autre. Un élément P est compatible avec un élément Q si P est catégorisé dans le même type que Q, ou dans un sous-type de Q.

```
ProcessElement::isCompatibleWith(p : ProcessElement) : Boolean;
isCompatibleWith = p->oclIsKindOf(self.oclType)
```

[1] Ces méta-opérations font partie du méta-modèle ; elles seront réutilisées plus tard dans plusieurs règles de bonne modélisation.

II.1.2. Relations entre éléments de Procédé

Le concept abstrait *ProcessRelation* est défini pour représenter une généralisation des relations entre éléments de procédé. La Figure II-15 montre les sous-classes de *ProcessRelation*.

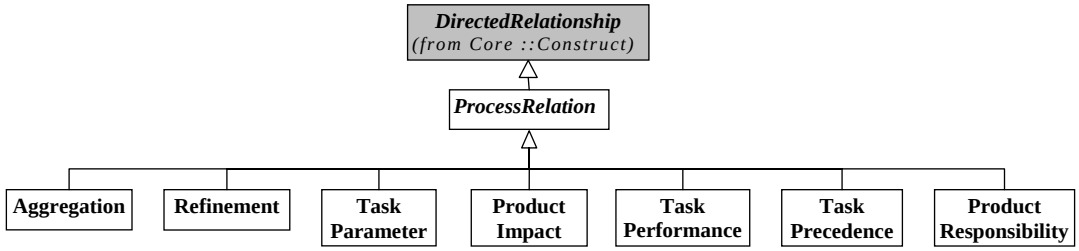


Figure II-15. Relations (entre éléments) de procédé

Nous définissons dans cette section les deux relations *Aggregation* et *Refinement* qui sont applicables pour tous les types d'éléments[1]. Les autres sous-classes sont décrites au fur et à mesure dans les sections suivantes.

Aggregation

Cette relation exprime un couplage entre deux éléments de même type. Dans cette relation, un élément (*composant*) fait partie de la composition structurelle ou fonctionnelle de l'autre (*agrégat*). La Figure II-16 montre la syntaxe abstraite de cette relation.

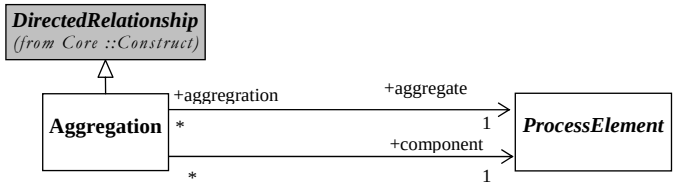


Figure II-16. Relation d'agrégation entre éléments compatibles

Refinement

Ce concept représente une dépendance entre deux éléments de même type dans laquelle la source (*refinedElement*) dérive de la cible (*originalElement*) en l'ajoutant de l'information ou la redéfinir. La Figure II-17 montre la syntaxe abstraite de cette relation.

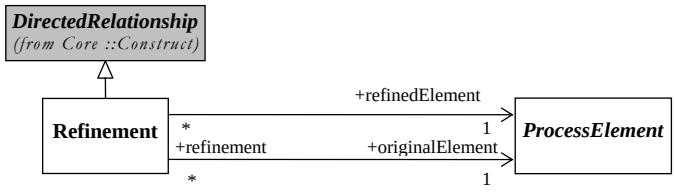


Figure II-17. Relation de raffinement entre éléments compatibles

[1] Chacune de ces deux relations est définie avec une sémantique générale qui sera spécialisée pour chaque type d'élément (produit, rôle, tâche). Nous décrirons donc la sémantique précise de ces relations dans le contexte spécifique de chaque type d'élément.

Opérations additionnelles

Les méta-opérations suivantes sont définies pour faciliter la définition des contraintes sur les relations *Aggregation* et *Refinement* :

[OP4] La méta-opération *getOriginalElements()* renvoie les éléments d'origine directs d'un élément raffiné.

```
ProcessElement::getOriginalElements(): Set(ProcessElement);
if self.refinement→notEmpty()
  getOriginalElements = self.refinement.originalElement
```

[OP5] La méta-opération *getallOriginalElements()* renvoie tous les éléments d'origine directs et indirects d'un élément raffiné.

```
ProcessElement::getallOriginalElements(): Set(ProcessElement);
if self.refinement→notEmpty()
getallOriginalElements = self.getOriginalElements()→
union(self.getOriginalElements→ collect(p|p.getallOriginalElements())
```

[OP6] La méta-opération *getRefinedElements()* renvoie les éléments directement raffinés depuis un élément.

```
ProcessElement::getRefinedElements(): Set(ProcessElement);
if self.refinement→ notEmpty()
  getRefinedElements = self.refinement.refinedElement
```

[OP7] La méta-opération *getComponents()* renvoie les éléments composants d'un élément d'agrégat.

```
ProcessElement::getComponents(): Set(ProcessElement);
if self.aggregation→notEmpty()
  getComponents = self.aggregation.component
```

Règles de bonne modélisation

Les contraintes suivantes sont définies pour préciser la sémantique ainsi que pour vérifier la cohérence des relations *Aggregation* et *Refinement* :

[C1] Dans une relation d'agrégation, le composant doit être compatible avec l'agrégat.

```
context Aggregation inv:
self.component.isCompatibleWith(self.aggregate))
```

[C2] Dans une relation de raffinement, l'élément raffiné doit être compatible avec l'élément originel.

```
context Refinement inv:
self.refinedElement.isCompatibleWith(self.originalElement))
```

[C3] L'élément originel d'une relation de raffinement doit être abstrait ou général.

```
context Refinement inv:
self.originalElement.getAbstractionLevel = 'Abstract' or 'General'
```

[C4] Si l'élément originel d'une relation de raffinement est un élément abstrait, l'élément raffiné doit être général ou concret.

```
context Refinement inv:
self.originalElement.getAbstractionLevel = 'Abstract' implies
self.refinedElement.getAbstractionLevel = 'General' or 'Concrete'
```

[C5] Un élément peut être directement raffiné depuis un seul autre élément.

```
context ProcessElement inv:
self.getOriginalElements → size()<=1
```

[C6] La relation de raffinement doit être acyclique.

```
context ProcessElement inv:
not self.getallOriginalElements()->includes(self)
```

Notation

La Tableau II-1 montre les notations des relations *Aggregation* et *Refinement*.

UML-PP Méta-classe	UML Classe de base	Notation	Notation alternative
Aggregation	Directed Relationship	**Agrégat** ◇— **Composant**	
Refinement	Directed Relationship	**Élément Originel** <-- « refine » -- **Élément Raffiné**	

Tableau II-1. Notations des relations d'agrégation et de raffinement

II.1.3. Produit

> Un produit (*Product*) est un artefact créé, consommé, ou modifié durant le processus de développement.

Nous classons les produits logiciels en trois niveaux d'abstraction : *abstrait*, *général* et *concret*. La Figure II-18 montre la typologie des produits selon leur niveau d'abstraction.

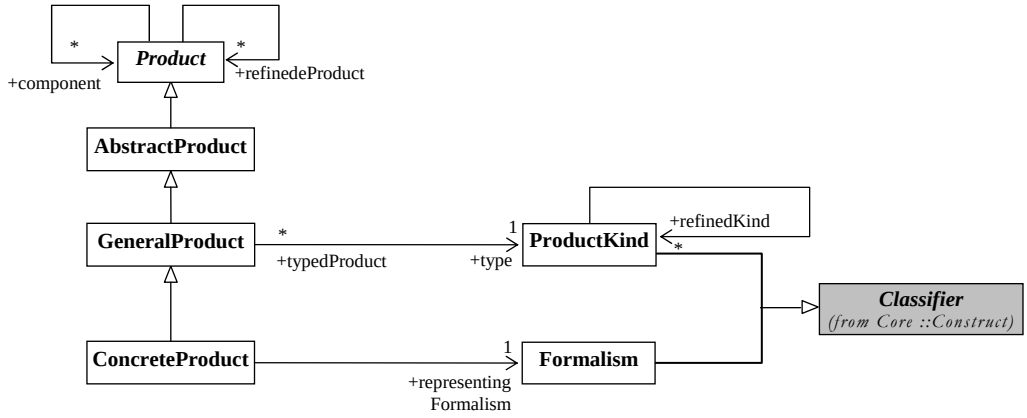


Figure II-18. Typologie de produits

AbstractProduct

Un produit abstrait *(AbstractProduct)* est utilisé pour décrire un produit quelconque. Il n'a aucun sens précis.

GeneralProduct

Un produit général (*GeneralProduct*) est un produit qui est spécifié par un type de produit (*ProductKind*). Un type de produit est définie par utilisateur et peut être spécialisé en plusieurs sous-types de produit pour des buts plus spécifiques.

Par exemple, Open Process Framework [OPF] définit les types de produits suivants : *Code, Modèle, Document, Application, Diagramme* et *Besoins*. Le type *Code* peut être spécialisé en trois types plus spécifiques, *Code orienté-objet, Code fonctionnel* et *Code logique*.

ConcreteProduct

Un produit concret est spécifié par un type de produit spécifique qui ne peut plus être spécialisé, et est représenté dans un formalisme concret (*Formalism*).

Un formalisme concret peut être un langage de description (par exemple *OMT, UML*) ou de programmation (par exemple *C, Java*) mais il peut être aussi une structure spécialisée pour représenter un type concret de produit (par exemple le template de *Spécification de Besoins* défini par *RUP* [Kruchten03]).

Étant des éléments de procédé, les produits peuvent participer aux relations *Aggregation* et *Refinement* définies dans la section II.1.2. Nous définissons la relation *ProductImpact* pour représenter la dépendance d'impact entre des produits. La participation d'un produit à l'exécution d'une tâche est exprimée par la relation *TaskParameter* ; la responsabilité d'un rôle sur l'élaboration ou la modification d'un produit est exprimée par la relation *ProductResponsibility*.

La Figure II-19 montre les relations définies sur les produits.

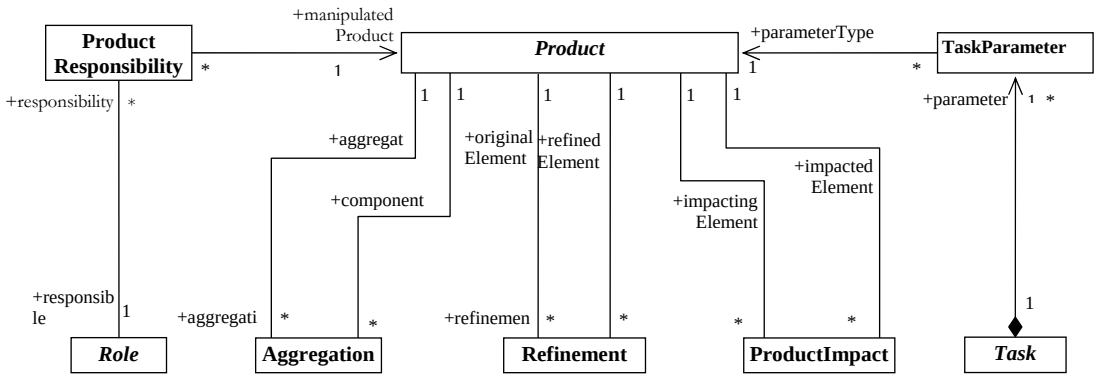


Figure II-19. Relations relatives aux produits

Nous présentons dans la suite la sémantique de ces relations.

Aggregation

Un produit (*aggregate*) peut être un agrégat de plusieurs produits composants (*component*). Dans le contexte de produits, une telle agrégation signifie que chaque produit composant définit une partie du contenu du produit agrégat. Un produit peut participer à plusieurs relations d'agrégation en tant qu'agrégat ou composant.

Par exemple, *Software Development Plan* est un produit qui est un agrégat de plusieurs autres produits, tels que *Project Organization*, *Schedule*, *Quality Plan*, *Test Plan*, *RiskPlan*, *Documentation Plan*.

Refinement

Un produit (*refinedElement*) est le raffinement d'un autre (*orignalElement*) si le type du premier est une spécialisation du type du second ; ou si le premier est un sous-type du second. Un produit peut être raffiné en plusieurs produits plus spécifiques, mais il ne peut être le raffinement que d'un seul autre produit.

Par exemple, un *diagramme* est un produit général de type *modèle*. Un modèle peut être un *modèle structurel* ou un *modèle comportemental*. Si l'on a un *diagramme de classe* (produit général) qui est un *modèle structurel*, on peut dire que le diagramme de classe est un raffinement de *diagramme*. Un *diagramme de classe de UML* (produit concret) est un raffinement de *diagramme de classe*.

ProductImpact

Une dépendance d'impact reliant un produit à un autre indique que la modification du premier (*impactingElement*) peut impacter l'autre (*impactedElement*). Un produit peut participer à plusieurs relations d'impact.

Par exemple, on utilise le *cahier des charges* pour éliciter des besoins d'utilisateurs et élaborer la *spécification des exigences*. Si le cahier des charges change, il faut remettre en cause la spécification des exigences. Il existe donc une dépendance *ProductImpact* entre le cahier des charges et la spécification des exigences.

Nous décrirons la relation *TaskParameter* dans le contexte du concept de tâche (c.f. section II.1.5). La relation *ProductResponsibility* sera décrite dans le contexte du concept de rôle (c. f. section II.1.4).

Règles de bonne modélisation

Les contraintes suivantes sont définies pour préciser la sémantique des concepts relatifs au produit, et pour assurer la cohérence des modèles contenant des produits :

[C7] Les composants d'un produit sont les produits référencés par ses relations d'agrégation.

```
context Product inv:
self.component = self.getComponents()
```

[C8] Les produits raffinés d'un produit sont les produits référencés par ses relations de raffinement.

```
context Product inv:
self.refinedProduct = self.getRefinedElements()
```

[C9] Un produit général B est un raffinement d'un produit général A si le type de produit de B est une spécialisation du type de produit de A.

```
context Refinement inv:
let A = self.originalElement
and  B = self.refinedElement in
(A.oclIsTypeOf(GeneralProduct)
(B.and B.oclIsTypeOf(GeneralProduct))implies
                A.type.refinedKind → includes (B.type))
```

[C10] Les composants d'un produit agrégat peuvent être plus concrets que leur agrégat, mais parmi ces composants il en faut un dont le niveau d'abstraction est le même que celui du produit agrégat.

```
context Product inv:
self.component → notEmpty() implies
     self.component → forAll(p|p.isCompatibleWith(self))and
                   exists(p|p.isIndenticalTo(self))
```

[C11] Dans une dépendance d'impact, le niveau d'abstraction du produit impactant peut être inférieur ou inférieur à celui du produit impacté.

```
context ProductImpact inv:
self.impactingElement.isCompatibleWith(self.impactedElement)
```

Notation

Nous adaptons la notation associée à *Classifier* pour représenter les produits, et celle de *Dependance* pour représenter la relation d'impact. Quant aux associations entre les sous-classes de *Product*, *ProductKind* et *Formalism* utilisées pour spécifier les produits à différents niveaux d'abstraction, nous décidons de les représenter comme des propriétés obligatoires de produits généraux ou concrets[1] afin de simplifier la représentation des produits.

Les notations proposées sont représentées dans le Tableau II-2.

[1] C'est-à-dire qu'en instanciant la méta-classe *GeneralProduct* pour définir un produit général, le concepteur doit spécifier l'attribut *ProductKind*. De la même façon, pour définir un produit concret, il doit spécifier les attributs *ProductKind* et *Formalism*.

Méta-classe UML-PP	Classe de base UML	Notation	Notation alternative
ProductKind	Classifier	«ProductKind» **Nom**	
Formalism	Classifier	«Formalism» **Nom**	
Abstract Product	Classifier	«AbstractProduct» **Nom**	AP **Nom**
General Product	Classifier	«GeneralProduct» **Nom** kind :ProductKind L'attribut *kind* reflète l'association entre la méta-classe *GeneralProduct* et la méta-classe *ProductKind* spécifiée dans le méta-modèle UML-PP (c.f.Figure II-18).	GP **Nom**
Concrete Product	Classifier	«ConcreteProduct» **Nom** kind: ProductKind format: Formalism L'attribut *format* reflète l'association entre la méta-classe *ConcreteProduct* et la méta-classe *Formalism* spécifiées dans le méta-modèle UML-PP (c.f. Figure II-18).	CP **Nom**
Product Impact	Directed Relationship	**Product** «impact » **Product**	

Tableau II-2. Notations des méta-classes de produits

Description d'un produit

La définition d'un produit comporte son type, le langage utilisé pour le représenter et sa composition, c'est-à-dire ses produits composants[1]. Ces derniers sont exprimés par des relations d'agrégation (*Aggregation*) qui les relient à l'agrégat. La description d'un produit peut refléter éventuellement les dépendances d'impacts (*ProductImpact*) concernant le produit défini.

Nous montrons dans la Figure II-20 des exemples de produits. Dans ces exemples, *P* est une représentation abstraite d'un produit quelconque. Un programme a pour type *Code* et est représenté par un produit général. Le type *Code* est raffiné en *OOCode* et *FunctionalCode,* donc un programme fonctionnel et un programme orienté-objet sont des spécialisations d'un programme et restent au niveau général. Par contre, un *JavaProgram* est spécifié par le type

[1] Certains de ces détails peuvent être omis selon le niveau d'abstraction du produit décrit.

OOCode et est codé en Java, un langage de programmation concret ; c'est donc un produit concret.

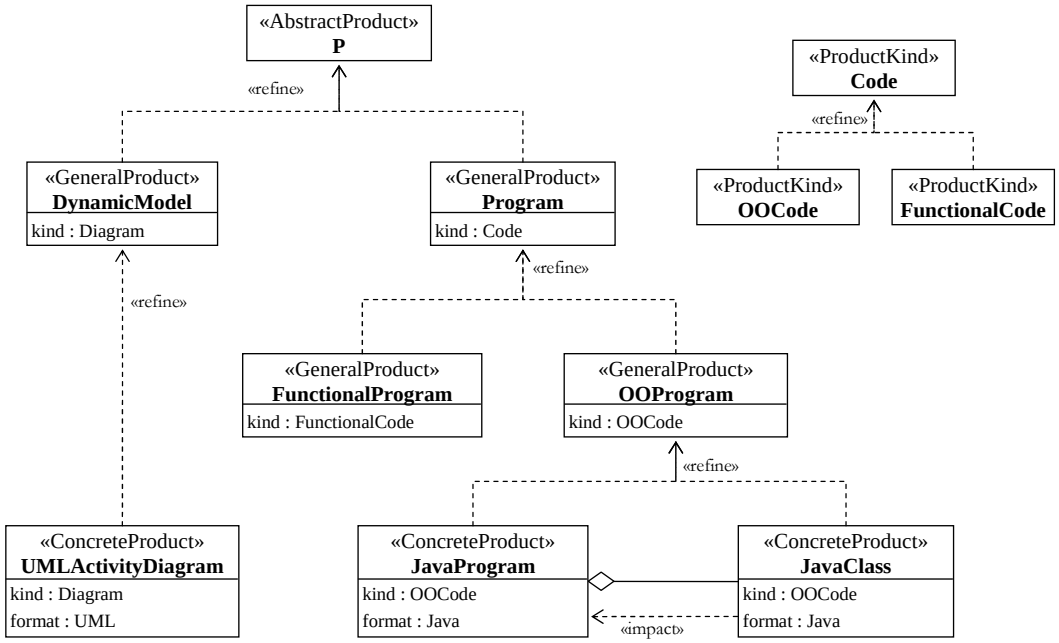


Figure II-20. Exemples de produits

II.1.4. Rôle

> Un rôle (*Role*) est un concept abstrait décrivant un ensemble de responsabilités et de compétences nécessaires pour effectuer certaines activités de développement.

Dans le contexte d'un projet d'ingénierie logicielle, un rôle définit la fonction d'un individu ou d'un ensemble d'individus travaillant en équipe (par exemple, le rôle *analyste*, le rôle *concepteur*). Il faut considérer que les rôles n'identifient pas des individus précis ; les membres individuels du projet peuvent jouer des rôles différents durant le projet. Nous classons également les rôles en trois niveaux d'abstraction : *abstrait, général* et *concret* (Figure II-21).

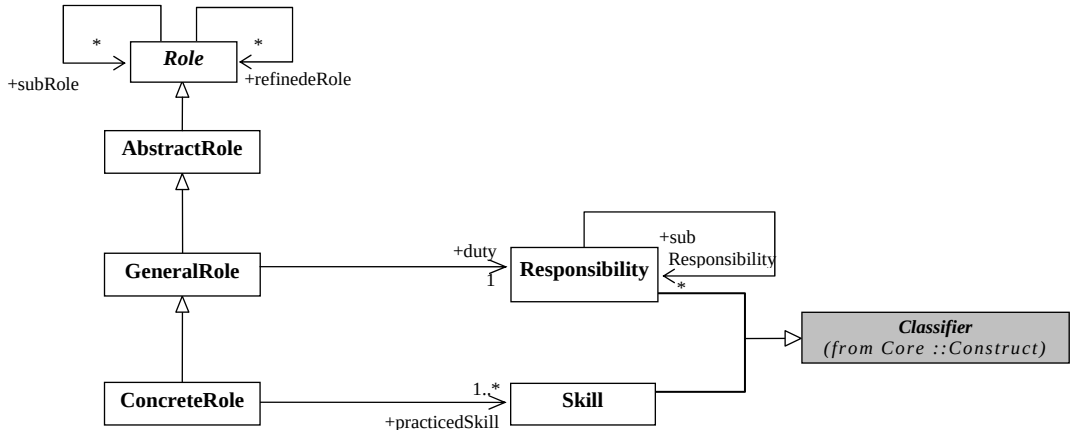


Figure II-21. Typologie de rôles

AbstractRole

Un rôle abstrait (*AbstractRole*) n'a aucune définition précise, ce concept est utilisé pour décrire un rôle quelconque.

GeneralRole

Un rôle général (*GeneralRole*) est associé à une responsabilité (*Responsability*). Une responsabilité spécifie une fonction de développement et peut être décomposée en plusieurs sous-responsabilités.

Par exemple, parmi les responsabilités générales demandées par le développement logiciel on peut citer *Analyse, Conception, Implémentation, Test, Gestion*.

ConcreteRole

Un rôle concret est spécifié par une responsabilité et des compétences (*Skill*) concrètes nécessaires pour accomplir la responsabilité. Les compétences spécifient les connaissances et l'expérience en méthodes et techniques de développement pour manipuler des produits concrets.

Par exemple, le rôle *concepteur* est associé à la responsabilité de concevoir le système. Pourtant, la conception d'un système en utilisant la méthode *OMT* requiert la capacité d'utilisation des notations OMT, alors que la conception dans l'approche RUP requiert l'utilisation de *UML*.

Comme les autre types d'élément de procédé, les rôles peuvent participer aux relations *Aggregation* et *Refinement* définies dans la section II.1.2. La relation *TaskPerformance* exprime la participation d'un rôle à l'exécution d'une tâche et la relation *ProductResponsibility* exprime la responsabilité d'un rôle dans l'élaboration ou la modification d'un produit.

La Figure II-22 montre les relations définies sur les rôles. Nous précisons dans la suite la sémantique de ces relations.

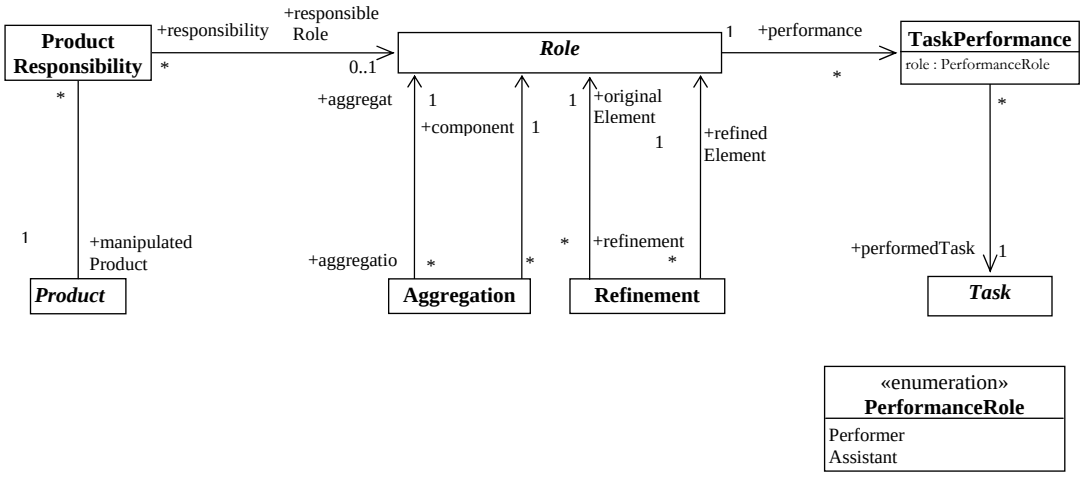


Figure II-22. Relations relatives aux rôles

Aggregation

Une relation d'agrégation reliant deux rôles signifie que le rôle du composant (*component*) prend en charge une partie de travail du rôle de l'agrégat (*aggregate)*. Autrement dit, une agrégation de rôle exprime que la responsabilité de l'agrégat est décomposée en plusieurs sous-responsabilités qui sont assignées aux rôles des composants.

Par exemple, le rôle *Tester* est un agrégat de trois rôles : *TestAnalyst*, *TestDesigner* et *TestConductor*.

Refinement

Un rôle (*refinedElement*) est le raffinement d'un autre (*orignalElement*) s'il est responsable de la même fonction que celle du second mais sur des produits plus spécifiques ; ou si le premier est un sous-type du second.

Par exemple, le rôle *Tester* peut être raffiné en *EmbeddedSystemTester, JavaSystemTester*.

ProductResponsibility

Cette relation établie entre un rôle et un produit exprime le fait que le rôle est responsable de la manipulation du produit.

Par exemple, le rôle *Test Manager* du procédé RUP [Kruchten03]est responsable de l'élaboration des produits *Test Plan* et *Test Evaluation Summary*.

TaskPerformance

Cette relation peut être établie entre un rôle et une tâche pour exprimer la participation du rôle à l'exécution de la tâche. La nature de la participation du rôle est déterminée par l'attribut *«role»* de la relation. Les valeurs possibles de l'attribut *«role»* sont définies par le type d'énumération *PerformanceRole* comme suit :

- *role = performer* : le rôle réalise la tâche
- *role = assistant* : le rôle assiste la tâche

Par exemple, dans le procédé RUP [Kruchten03], le rôle *System Analyst* réalise la tâche *Elicit Stakeholders Requests* et assiste la tâche *Review Requirements*.

Règles de bonne modélisation

Les contraintes suivantes sont définies pour préciser la sémantique des concepts relatifs au rôle, et pour assurer la cohérence des modèles contenant des rôles :

[C12] Les sous-rôles d'un rôle sont les rôles référencés par ses relations d'agrégation.

```
context Role inv: self.subRole = self.getComponents()
```

[C13] Un rôle A est un sous-rôle d'un rôle B signifie que la responsabilité de A est une sous-responsabilité de la responsabilité de B.

```
context Role inv: let A,B:Role in
B.subRole→includes(A)implies B.duty.subResonsibility→includes(A.duty)
```

[C14] Les rôles raffinés d'un rôle sont les rôles référencés par ses relations de raffinement.

```
context Role inv: self.refinedRole = self.getRefinedElemens()
```

[C15] Un rôle A est un raffinement d'un rôle B s'ils ont la même responsabilité et que les produits manipulés par A sont les raffinements de ceux manipulés par B.

```
context Role inv:  let A,B:Role
let Aproduct = A.responsibility.manipulatedProduct
let BProduct = B.responsibility.manipulatedProduct in
B.refinedRoles → includes(A)implies
(A.duty=B.duty)and (Bproduct.getRefinedElements())→includes(Aproduct))
```

[C16] Les sous-rôles d'un rôle agrégat peuvent être plus concrets que leur agrégat, mais parmi eux il en faut un dont le niveau d'abstraction est le même que celui du rôle agrégat.

```
context Rôle inv:
self.component → notEmpty() implies
     self.component → forAll(r|r.isCompatibleWith(self))and
                    exists(r|r.isIndenticalTo(self))
               self.component.isIdenticalTo(self.aggregate))
```

[C17] Il ne peut pas y avoir plus d'une relation *TaskPerformance* entre un rôle et tâche (c'est-à-dire qu'un rôle ne peut pas réaliser et assister une même tâche à la fois).

```
context Role inv:
self.performance → forAll(p1,p2|p1.performedTask<>p2.performedTask)
```

[C18] Il y a une seule relation *ProductResponsibility* entre un rôle et un produit.

```
context Role inv:
self.responsibility →
     forAll(r1,r2|r1.manipulatedproduct<>r2.manipulatedproduct)
```

[C19] Un rôle qui est responsable d'un ensemble de produits doit réaliser les tâches manipulant ces produits.

```
context Role inv:
self.responsibility.manipulatedproduct → notEmpty() implies
self.Performance.performedtask.parameter →
             select(p|p.kind=#out or p.kind=#inout)
                                 = self.responsibility.manipulatedproduct
```

Notation

Comme pour la définition de la syntaxe concrète pour les produits, nous adaptons la notation de *Classifier* pour représenter les rôles, et la notation de *DirectedRelationship* pour représenter les relations relatives aux rôles. Les responsabilités et les compétences de rôles sont aussi représentées comme des propriétés obligatoires des rôles généraux et concrets.

Le Tableau II-3 montrent les notations proposées pour modéliser les rôles.

UML-PP Méta-classe	UML Classe de base	Notation	Notation alternative
Responsibility	Classifier	«Responsibility» **Nom**	
Skill	Classifier	«Skill» **Nom**	
Product Responsibility	Directed Relationship	**Role** «responsible for » **Product**	
Task Performance	Directed Relationship	**Role** « perform » {role} **Task** {*role*} spécifie la nature de la participation du rôle à l'exécution de la tâche : *performer* ou *assistant*.	
AbstractRole	Classifier	«AbstractRole» **Nom**	AR **Nom**
GeneralRole	Classifier	«GeneralRole» **Nom** duty :Responsibility L'attribut *duty* traduit l'association entre la méta-classe *GeneralRole* et la méta-classe *Responsability* spécifiée dans le méta-modèle UML-PP (c.f. Figure II-21).	GR **Nom**
ConcreteRole	Classifier	«ConcreteRole» **Nom** duty: Responsibility competence :Skill [1..*] L'attribut *competence* traduit l'association entre la méta-classe *ConcreteRole* et la méta-classe *Skill* spécifiée dans le méta-modèle UML-PP (c.f. Figure II-21).	CR **Nom**

Tableau II-3. Notations des méta-classes de rôles

Description d'un rôle

La définition d'un rôle décrit sa responsabilité, y compris les tâches qu'il doit réaliser ou assister , et les produits dont il est responsable. Cela est représenté par des relations de performance (*TaskPerformance*) et des relations de responsabilité de produits *(ProductResponsibility)* auxquelles le rôle participe. La définition d'un rôle peut éventuellement

décrire ses sous-rôles en utilisant des relations d'agrégation de rôle. Les compétences concrètes du rôle peuvent être représentées ou non selon son niveau d'abstraction.

La Figure II-23 montre des exemples de rôles extraits du procédé RUP [Kruchten03].

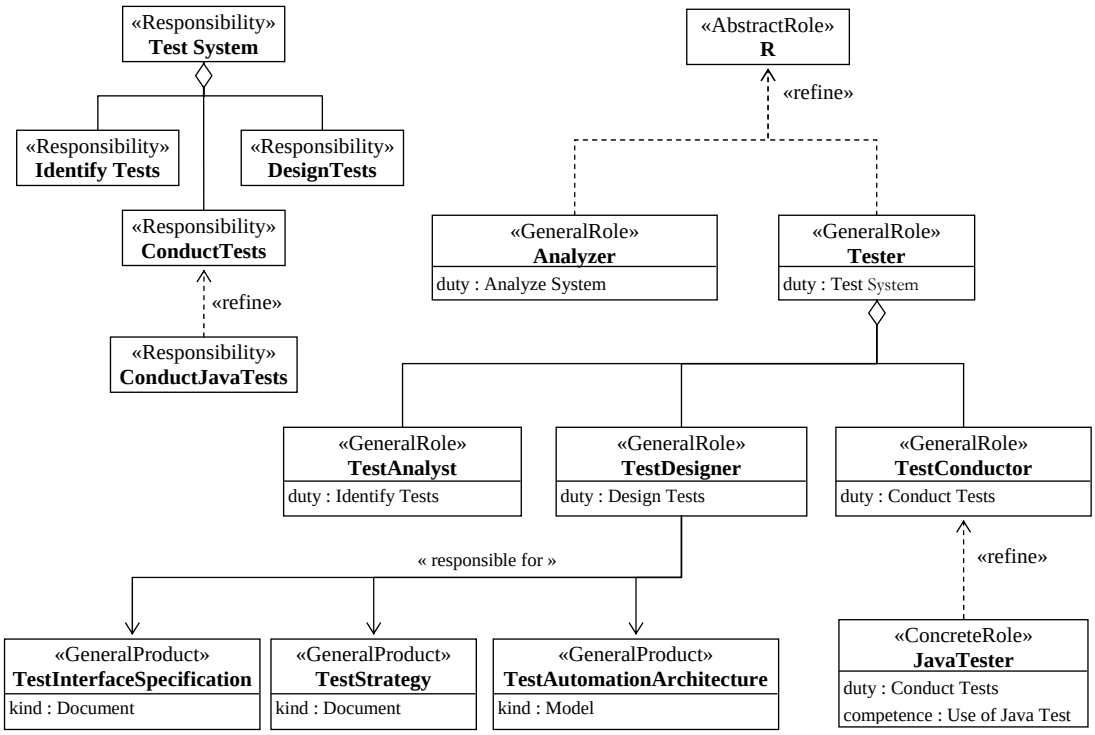


Figure II-23. Exemple de rôles

Dans ces exemples, la responsabilité *TestSystem* du rôle général *Tester* est raffinée en responsabilités plus spécifiques (*Indentify Required Tests*, *Design Tests* et *Conduct Tests*) assignées respectivement aux rôles plus spécifiques *TestAnalyst*, *TestDesigner* et *TestConductor*. Quant au rôle *JavaTester*, concret, il requiert la capacité d'utiliser des outils spécialisés pour vérifier du code Java.

II.1.5. Tâche

> Une tâche (*Task*) est une unité de travail contrôlée et réalisée dans le but de créer ou modifier un (des) produit(s).

Une tâche est une unité de travail contrôlée d'un procédé, c'est-à-dire qu'elle est assignée à un participant du processus et son exécution est surveillée durant le déroulement d'un projet. D'un point de vue structurel, une tâche peut être décomposée en sous-tâches. D'un point de vue comportemental, elle peut se décomposer en actions (*Step*) (c.f. la section II.1.6 pour la définition du concept *Step*).

Parce que les actions précises d'une tâche dépendent des produits manipulés, nous proposons une catégorisation des tâches selon les produits qu'elles manipulent. Comme pour les autres types d'éléments de procédé, nous classons également les tâches en trois niveaux d'abstraction : *abstrait, général* et *concret*.

La Figure II-24 montre la hiérarchie des tâches.

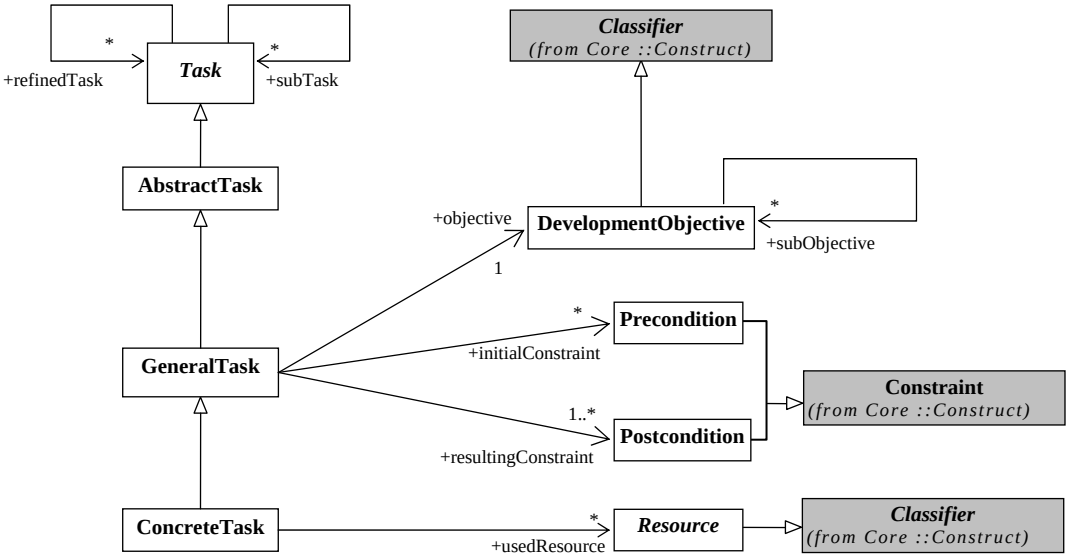


Figure II-24. Typologie de tâches

AbstractTask

Une tâche abstraite (*AbstractTask*) n'a aucune sémantique associée, c'est-à-dire qu'elle n'exprime aucun objectif de développement. Elle est juste une détentrice de place *(«place holder»)* pour une tâche quelconque.

Les produits manipulés par une tâche abstraite peuvent être abstraits[1]. Une tâche abstraite n'est pas exécutable.

GeneralTask

Une tâche générale (*GeneralTask*) possède un but de développement (*DevelopmentObjective*) concernant la création ou la modification de produits généraux. Elle peut être décomposée en actions à réaliser pour accomplir son but. On peut spécifier également des conditions pour déclencher ou terminer une tâche générale (*PreCondition* et *PostCondition*). Une tâche générale ne peut pourtant pas être exécutée, car sa mise en oeuvre (c'est-à-dire les détails de réalisation de ses actions) dépend de produits incomplètement spécifiés.

Les produits manipulés par une tâche générale doivent comprendre au moins un produit général, mais pas de produit abstrait.

ConcreteTask

Une tâche concrète (*ConcreteTask*) a pour but de créer ou modifier des produits concrets. Par conséquent, elle peut être décomposée en actions complètement décrites en termes de ressources utilisées (outils, standards, etc.). Une tâche concrète est donc exécutable.

---

[1] En effet, on peut avoir une tâche abstraite qui ne manipule aucun produit abstrait. Par exemple, pour décrire une action quelconque sur une liste d'objets, on peut utiliser la tâche *Dosomething*. Dans ce cas, *Dosomething* n'as aucune sémantique et reste abstraite, alors que la liste qu'elle manipule est un produit général.

Les produits manipulés par une tâche concrète ne comportent que des produits concrets.

Dans notre catégorisation, le niveau d'abstraction d'une tâche dépend du niveau d'abstraction des produits qu'elle traite. Si une tâche travaille sur plusieurs produits ayant des niveaux d'abstraction différents, le niveau d'abstraction de la tâche est déterminé par le niveau d'abstraction le plus élevé de ses produits.

Comme les produits et les rôles, les tâches peuvent participer à des relations *Aggregation* et *Refinement* pour exprimer respectivement leurs compositions et leurs raffinements. Les produits manipulés par une tâche sont explicités par la relation *TaskParameter*. La relation *TaskPrecedence* représente la dépendance d'ordre d'exécution entre des tâches. Finalement, les rôles participant à l'exécution d'une tâche sont spécifiés en utilisant la relation *TaskPerformance* qui a été présentée dans le contexte du concept de rôle (c.f. II.1.4).

La Figure II-25 montre les relations définies sur les tâches. Nous précisons dans la suite la sémantique de ces relations.

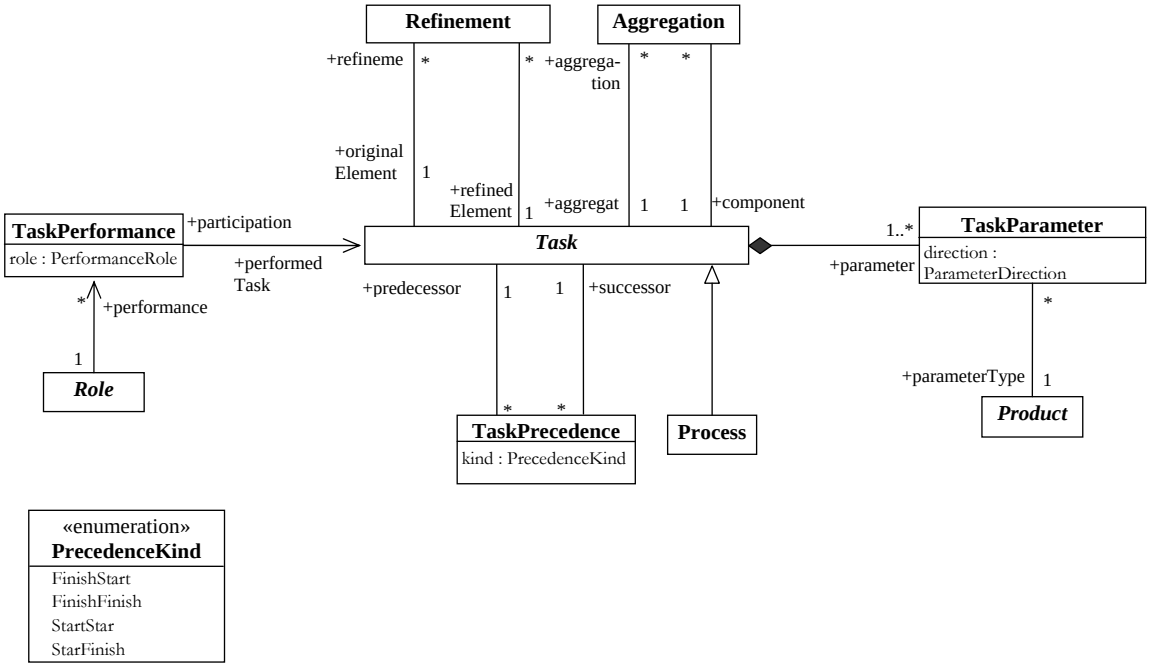


Figure II-25. Relations relatives aux tâches

Aggregation

Une agrégation de tâche signifie que chaque sous-tâche (*component*) réalise une partie du travail de la tâche *agrégat*. Autrement dit, les tâches composantes doivent être réalisées ensemble pour accomplir l'objectif de développement de la tâche supérieure.

Par exemple, la tâche *Review a Code* a trois sous-tâches : *Detect Code Errors, Modify Code* et *Evaluate Code*.

Nous considérons un procédé (*Process*) comme une tâche racine qui est l'agrégat (direct ou indirect) de toutes les autres tâches.

Refinement

Une tâche raffinée (*refinedElement*) a le même but de développement que la tâche originelle (*orignalElement*) mais elle travaille sur des produits plus spécifiques que ceux de la tâche originelle.

Par exemple, la tâche *Review a JavaCode* est un raffinement de la tâche *Review a Code*.

TaskParameter

Cette relation permet d'expliciter les produits manipulés par une tâche. Dans cette relation, l'attribut «*direction*» indique la nature d'un paramètre, qui peut être un produit d'entrée, de sortie ou les deux. L'attribut «*optional*», quant à lui, permet de spécifier si un paramètre est indispensable (*optional* = False) ou non. Un paramètre optionnel peut être omis au cours de l'exécution de la tâche.

Par exemple, pour la tâche *Review a Code,* le code à réviser est un paramètre requis en tant que produit d'entrée et de sortie , alors que le rapport de révision est un paramètre de sortie.

TaskPerformance

Cette relation exprime la participation d'un rôle à l'exécution d'une tâche. Une tâche peut être réalisée par un rôle et assistée par plusieurs rôles (c.f. II.1.4)

TaskPrecedence

Cette relation représente une dépendance entre deux tâches dans laquelle l'exécution d'une tâche dépend du début ou de la terminaison de l'autre.

Le type de dépendance est spécifié par l'attribut «*kind*» qui peut avoir une des valeurs suivantes :

- *FinishStart* (FS): le successeur doit attendre la terminaison du prédécesseur pour commencer.
- *FinishFinish*(FF) : le successeur doit attendre la terminaison du prédécesseur pour terminer.
- *StartFinish*(SF) : le successeur doit attendre le début du prédécesseur pour terminer.
- *StartStart* (SS) : le successeur doit attendre le début du prédécesseur pour commencer.

Par exemple, la tâche *Review Requirements* ne peut commencer que quand la tâche *Elicit Requirements* est finie. Il existe donc une dépendance *TaskPrecedence* de type FS entre elles. Entre la tâche *Change Control* et les tâches *Elicit Requirements*, *Analysis&Design*, *Construc*t et *Test*, en revanche, il existe des dépendances *TaskPrecedence* de type FF.

Règles de bonne modélisation

Les contraintes suivantes sont définies pour renforcer la sémantique des tâches :

[C20] Chaque tâche doit avoir au moins un paramètre représentant un produit en sortie ou en entrée-sortie.

```
context Task inv:
self.parameter → exists(p|p.direction=ParameterDirection::Out
                                or p.kind= ParameterDirection::InOut)
```

[C21] Les paramètres sortants d'une tâche abstraite doivent comprendre au moins un produit abstrait.

```
context Task inv:
if self.oclIsTypeOf(AbstractTask)then
            self.parameter → exist(oclIsTypeOf(AbstractProduct))
```

[C22] Les paramètres sortants d'une tâche générale doivent comprendre au moins un produit général, mais aucun produit abstrait.

```
context Task inv:
if self.oclIsTypeOf(GeneralTask)then
 self.parameter → excludes(oclIsTypeOf(AbstractProduct))
       and exists(oclIsTypeOf(GeneralProduct))
```

[C23] Tous les paramètres d'une tâche concrète doivent être des produits concrets.

```
context Task inv:
if self.oclIsTypeOf(ConcreteTask)then
 self.parameter → forAll(oclIsTypeOf(ConcreteProduct))
```

[C24] Une tâche est réalisée par un seul rôle.

```
context Task inv:
self.participation → select(p| p.role =Performer)→size()<=1
```

[C25] Les sous-tâches d'une tâche sont les tâches référencées par ses relations d'agrégation.

```
context Task inv:
self.subTask = self.getComponents()
```

[C26] Si une tâche A est une sous-tâche d'une tâche B, cela signifie que l'objectif de A est un sous-objectif de celui de B.

```
context Task inv: let A,B:Role in
B.subTask → includes(A)implies
B.objective.subObjective → includes(A.objective)
```

[C27] Les tâches raffinées d'une tâche sont les tâches référencées par ses relations de raffinement.

```
context Task inv: self.refinedTask = self.getRefinedElemens()
```

[C28] Une tâche A est un raffinement d'une tâche B si elles ont le même objectif et que les produits manipulés par A sont des raffinements de ceux manipulés par B.

```
context Task inv:
let A,B:Task
let Aproduct = A.parameter.parameterType
let BProduct = B.parameter.parameterType  in
B.refinedTask → includes(A)implies
          (A.objective=B.objective) and
          (Bproduct.getRefinedElements()→includes(Aproduct))
```

[C29] Les sous-tâches d'une tâche peuvent être plus concrètes que leur tâche supérieure, mais parmi elles il en faut une dont le niveau d'abstraction est le même que celui de la tâche supérieure.

```
context Task inv:
self.subTask → notEmpty() implies
     self.subTask → forAll(t|t.isCompatibleWith(self))and
                    exists(t|t.isIndenticalTo(self))
```

[C30] S'il existe une dépendance *TaskPrecedence* entre deux tâches, les produits de sortie du successeur ne peuvent pas être des produits d'entrée du prédécesseur.

```
context TaskPrecedence inv:
let input   = self.predecessor.parameter
let output  = self.successor.parameter
        input → excludes(output)
```

Description d'une tâche

La définition d'une tâche décrit son objectif de développement, les rôles participant à son exécution et les produits qu'elle manipule. Cela est représenté par des relations d'exécution (*TaskPerformance*) et des relations de paramétrage (*TaskParameter*) auxquelles le rôle participe.

La définition d'une tâche décrit également ses sous-tâches en utilisant des relations d'agrégation de tâches.

Les autres détails de la tâche comme les ressources utilisées, les actions primitives peuvent être représentés ou non selon le niveau d'abstraction.

Notation

Pour pouvoir décrire l'aspect dynamique des procédés avec les diagrammes d'activité de UML 2.0, nous adaptons la notation d'*Activity* de UML 2.0 pour représenter le concept de tâche. En utilisant la notation d'*Activity*, les relations entre une tâche et son objectif de développement, ses contraintes, ses paramètres et ses ressources sont représentés implicitement comme attributs de l'activité.

Le Tableau II-4 montre les notations proposées pour modéliser les tâches.

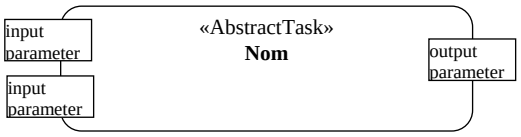
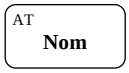
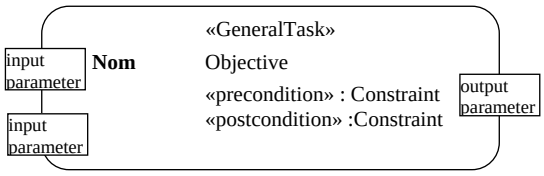
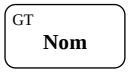
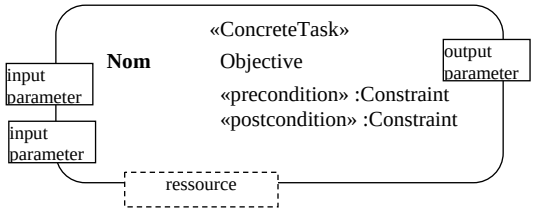
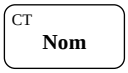
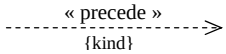
UML-PP Méta-classe	**UML Classe de base**	**Notation**	**Notation alternative**
Abstract Task	Classifier		
General Task	Classifier	 Les attribut *Objective*, *precondition* et *postcondition* reflètent respectivement les associations entre la méta-classe *GeneralTask* et les méta-classes *DevelopmentObjective*, *Precondition* et *Postcondition* spécifiées dans le méta-modèle UML-PP. (c.f. Figure II-24) Les paramètres sont représentés en bordures gauche et droite de la tâche.	
Concrete Task	Classifier	 Les ressources sont représentées en bordure inférieure de la tâche.	
Task Precedence	Directed Relationship	 {*kind*} spécifie la valeur de l'attribut *kind* de la dépendance *TaskPrecedence* (FS,FF,SF,SS)	

Tableau II-4. Notations des méta-classes relatives aux tâches

Nous donnons des exemples de types de tâches dans la Figure II-26.

Dans le cas **(a)**, nous montrons une tâche abstraite nommée *T* ayant un paramètre *P* typé comme produit abstrait. Le paramètre *P* représente un produit modifié par la tâche *T*.

Le cas **(b)** illustre la tâche générale *Review a Code* pour réviser un code quelconque. C'est une tâche générale, car son paramètre est un produit général (*Code*). On peut décomposer cette tâche en trois sous-tâches également générales : *DetectError*, *ModifyCode* et *EvaluateCode*.

Le cas **(c)** illustre la tâche concrète *Review a JavaCode* pour réviser un code Java ; elle est donc un raffinement de la tâche générale *Review a Code*. En connaissant le langage concret du code (Java) on peut fournir les détails d'implémentation de cette tâche. Par exemple la liste

d'erreurs typiques en Java (*JavaErrorChecklist*) [OMII04] et l'outil *AppPerfectCodeAnalyzer* [APPPerfect02] sont utilisés pour vérifier un *JavaCode.*

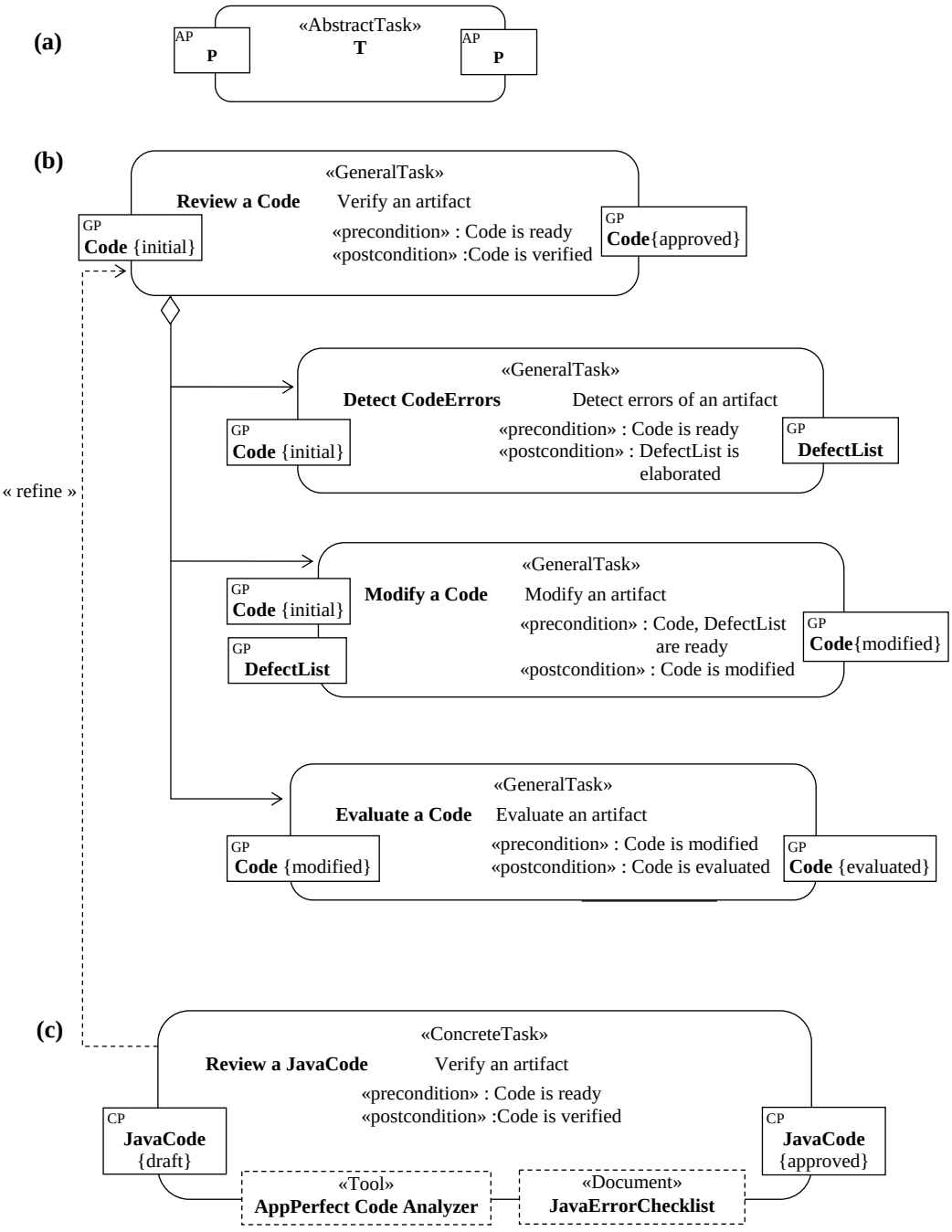


Figure II-26. Exemples de tâches

Nous montrons dans la Figure II-27 des exemples de dépendance de précédence entre tâches.

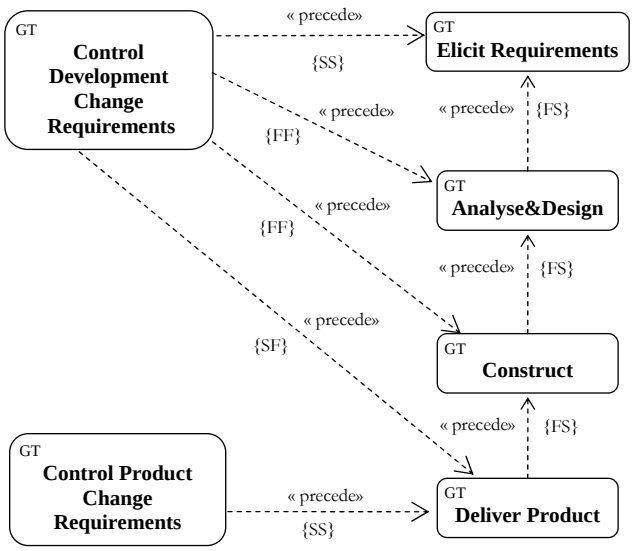


Figure II-27. Exemples de précédences entre tâches

Dans ces exemples, les tâches *Elicit Requirements*, *Analyse&Desgin*, *Construct* et *DeliverProduct* sont des étapes séquentielles du procédé. Il existe donc entre elles des dépendances *TaskPrecedence* de type Finish-Start. La tâche *Control Development Change Requirements* ne peut commencer que quand la tâche *Elicit Requirements* a commencé. Il y a donc une dépendance de type Start-Start entre elles. Par contre, la tâche *Control Development Change* doit durer pendant *Analyse&Desgin* et *Construct* pour surveiller les demandes de changement. Il y a donc des dépendances de type Finish-Finish entre elles. Une fois la tâche *Deliver Product* commencée, la tâche *Control Development Change* peut se terminer, donc il y a entre elles une dépendance de type Start-Finish.

II.1.6. Modèle de Procédé

Nous introduisons le concept de modèle de procédé (*ProcessModel*) pour représenter un fragment de procédé ou le contenu d'un élément de procédé.

Un modèle de procédé est composé de nœuds (*Node*) et d'arcs (*Edge*) qui relient les nœuds dans le modèle. Les nœuds sont des éléments de procédé et les arcs sont des relations définies entre des éléments de procédé.

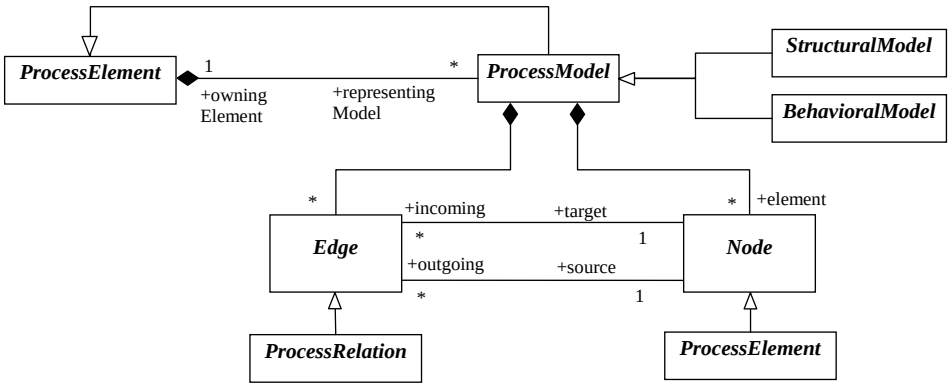


Figure II-28. Méta-modèle d'un modèle de procédé

Comme nous l'avons dit dans la section I.2.2, un modèle peut représenter l'aspect structurel ou comportemental d'un (fragment de procédé) procédé. Nous définissons donc deux types de modèles de procédé :

StructuralModel

Un modèle structurel décrit la composition d'un élément de procédé ou des relations organisationnelles entre éléments de procédé.

La Figure II-29 montre la syntaxe abstraite d'un modèle structurel de procédé dans notre méta-modèle.

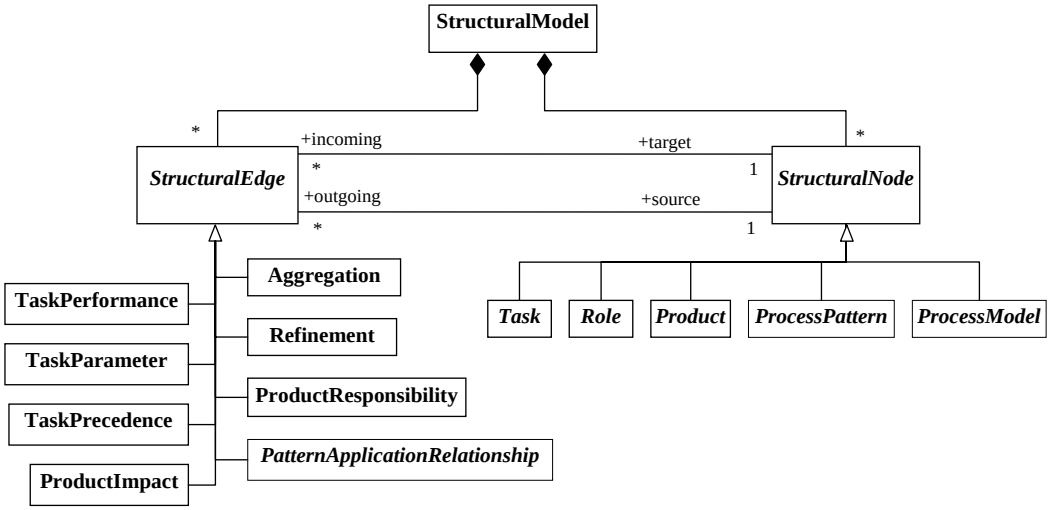


Figure II-29. Méta-modèle d'un modèle structurel de procédé

Pour permettre de représenter les patrons de procédé dans les modèles structurels, nous considérons les patrons ainsi que les modèles eux-mêmes comme des nœuds structurels. Les relations définies sur l'application des patrons sont considérées comme des arcs structurels. La Figure II-30 montre un exemple de modèle structurel.

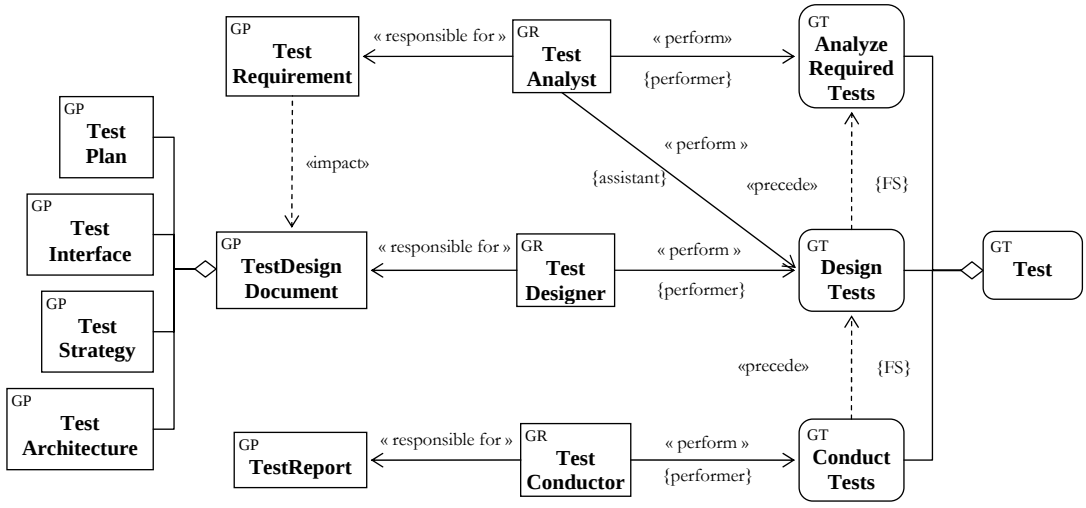


Figure II-30. Exemple de modèle structurel du procédé *Test*

Le modèle structurel dans cet exemple décrit la structure de la tâche *Test* qui comporte trois sous-tâches : *Analyze RequiredTests*, *Design Tests* et *Conduct Tests*. Le modèle montre les rôles qui réalisent ces sous-tâches, ainsi que les produits dont ces rôles sont responsables. Le modèle décrit également les dépendances de précédence entre sous-tâches, les structures des produits (par exemple le produit *Test Design Document* est un agrégat de quatre produits : *Test Plan*, *Test Interface*, *Test Strategy* et *Test Architecture*) et les dépendances entre produits (par exemple la dépendance d'impact entre les produits *Test Requirement* et *Test Design Document*).

BehavioralModel

Un modèle comportemental montre l'aspect dynamique d'un élément de procédé ou des relations de coordination entre éléments de procédé. Dans notre méta-modèle, nous utilisons ce type de modèle pour représenter le flux d'actions d'une tâche ou l'enchaînement des tâches.

La Figure II-31 montre la syntaxe abstraite d'un modèle comportemental de procédé. Cette syntaxe est adaptée de la syntaxe du diagramme d'activité de UML 2.0 SuperStructure [UML05b]. Les concepts originels d'UML sont repérés par des éléments sur fond gris.

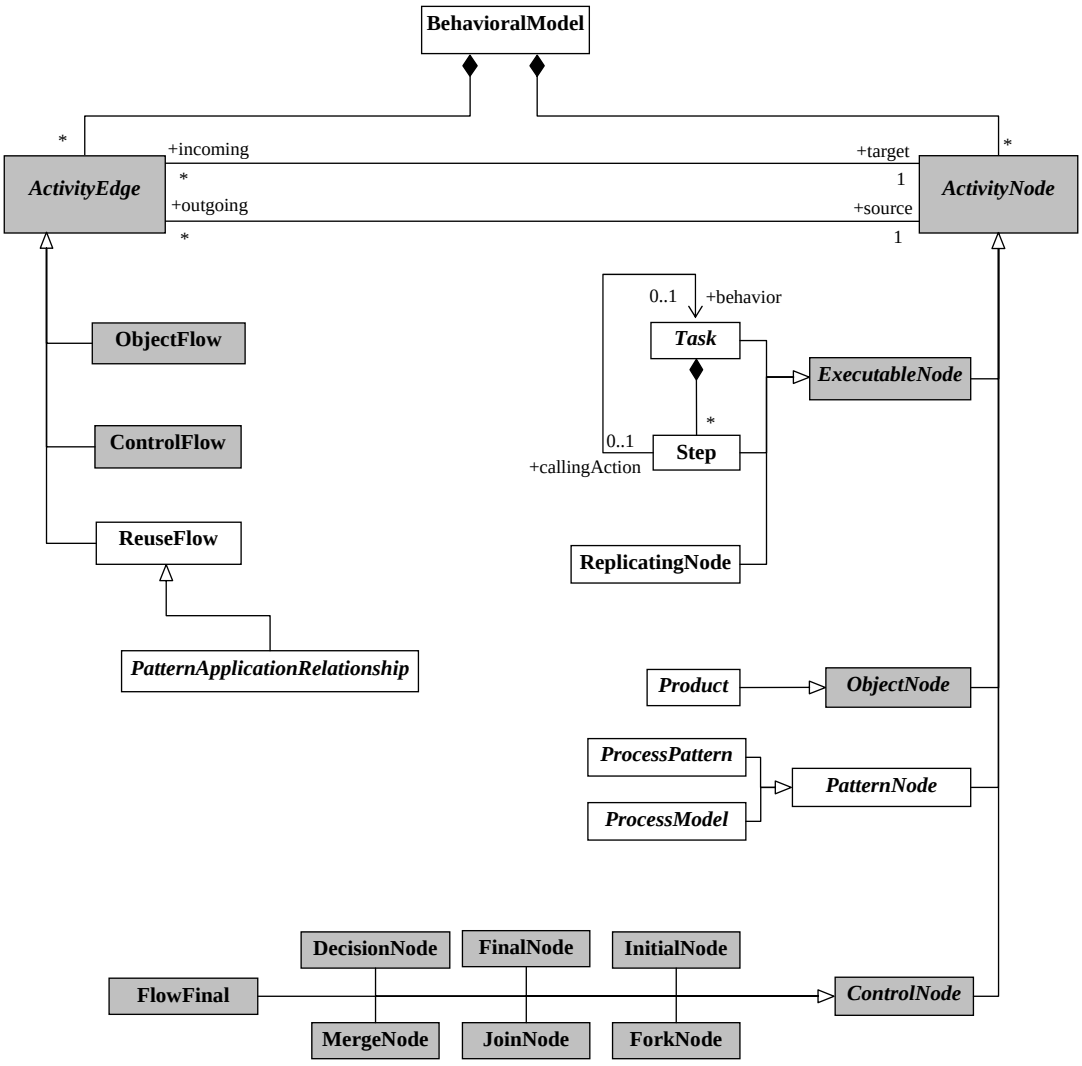


Figure II-31. Méta-modèle d'un modèle comportemental de procédé

Dans la suite nous décrivons brièvement la sémantique des concepts UML dans le contexte de notre méta-modèle. Pour des raisons de simplicité, nous ne présentons pas ici tous les détails de la définition du concept d'activité de UML. Nous mettons l'accent plutôt sur les nouveaux concepts que nous ajoutons (ou adaptons) pour la représentation de procédés et pour permettre la représentation de patrons de procédé.

ActivityNode

Un nœud d'activité est un type d'élément abstrait permettant de représenter les étapes le long du flux d'une tâche. UML définit trois types de nœuds d'activité: *ExecutableNode*, *ObjectNode* et *ControlNode*. Nous ajoutons un nouveau type de nœud d'activité, le *PatternNode*.

- **ExecutableNode**

  Un nœud exécutable est un nœud d'activité qu'on peut exécuter. Nous considérons les deux concepts suivants comme des nœuds exécutables :

  Step

  Un *Step* est une étape discrète (action) dans l'exécution d'une tâche. Autrement dit, les *Steps* sont des actions primitives qui participent au comportement d'une tâche. L'exécution d'un *Step* représente une transformation dans le processus modélisé. Un *Step* peut être une action de création, modification ou destruction de produit. En particulier, un *Step* peut invoquer une tâche (similaire au concept de *CallBehaviorAction* de UML). Ce mécanisme permet la réutilisation de tâches ainsi que l'imbrication de tâches.

  Quand on utilise un modèle comportemental pour décrire le comportement d'une tâche, les nœuds exécutables doivent être des *Steps*.

  Task

  Comme nous l'avons déjà défini, une tâche est une unité de travail contrôlée, ayant pour but de créer ou modifier des produits. Son comportement peut se décomposer en plusieurs *Steps*. Un *Step* peut à son tour invoquer une tâche (à travers le rôle behavior de l'association) ce qui permet de décrire la composition de tâches.

  Pour pouvoir décrire l'enchaînement des tâches par un modèle comportemental indépendant (non attaché à une tâche), nous définissons également une tâche comme un nœud exécutable.

  ReplicatingNode

  Nous proposons ce type spécial pour représenter une réplique comportementale multiple dans un modèle. Plus précisément, un tel nœud représente une suite de nœuds d'activité (nombre de nœuds non déterminé) qui respectent le comportement défini par les nœuds précédents.

- **ObjectNode**

  Un nœud d'objet représente un objet manipulé par une action dans une activité. Dans notre méta-modèle, un nœud d'objet représente un produit (*Product*).

- **ControlNode**

  Un nœud de contrôle est un nœud d'activité abstrait utilisé pour coordonner les flots entre les nœuds d'une activité. Nous réutilisons les types de nœud de contrôle définis par UML : nœud initial (*InitialNode*), nœud de fin d'activité (*FinalNode*), nœud de fin de flot (*FlowFinal*), nœud de décision (*DecisionNode)*, nœud de fusion (*MergeNode*), nœud de bifurcation (*ForkNode*), nœud d'union (*JoinNode*).

- **PatternNode**

  C'est un nouveau concept proposé pour permettre la représentation des applications des patrons de procédé dans un modèle comportemental. Un nœud de patron peut représenter un patron de procédé (*ProcessPattern*) ou un modèle de procédé (*ProcessModel*). Nous décrivons le concept de patron de procédé et ses applications dans les sections II.2 et II.3.

ActivityEdge

Un arc d'activité est un type d'élément abstrait permettant de représenter les liens entre les nœuds d'activités. UML définit deux types d'arc d'activités : *ControlFlow* et *ObjectFlow*. Nous ajoutons un nouveau type d'arc d'activités, le *ReuseFlow*.

- **ControlFlow**

  Un flot de contrôle représente le passage d'un nœud exécutable ou d'un nœud de contrôle vers un autre. Ce type d'arc spécifie l'enchaînement des actions ou des tâches.

- **ObjectFlow**

  Un flot d'objets représente les données d'entrée ou de sortie d'un nœud exécutable ; il représente également le passage des données entre des nœuds exécutables.

- **ReuseFlow**

  Nous proposons ce nouveau type d'arc d'activité pour relier les nœuds de patron avec des nœuds exécutables. C'est un type d'élément abstrait permettant de représenter les relations d'application de patrons de procédé qui seront décrites dans la section II.3.

Règles de bonne modélisation

Afin d'assurer la cohérence sémantique d'un modèle de procédé, nous définissons les contraintes suivantes :

[C31] Si un modèle de procédé est attaché à un élément de procédé, les éléments contenus dans le modèle sont les composants de l'élément possédant le modèle

```
context ProcessModel inv:
if self.owningElement → notEmpty() implies
self.owningElement.getComponents()→
          includes(self.element→select(e| oclIsKindOf(ProcessElement)))
```

[C32] Dans un modèle comportemental décrivant le comportement d'une tâche, les nœuds exécutables doivent être des *Steps*.

```
context BehavioralModel inv:
if self.owningElement → notEmpty() implies
self.element→ select(e|e.oclIsTypeOf(ExecutableNode))→
                                                forAll(e|e.ocsIsTypeOf(Step)
```

Notation

Le Tableau II-5 montre les notations proposées pour modéliser les concepts d'un modèle comportemental. Dans ce tableau, nous ne représentons que les notations des nouveaux concepts de UML-PP. Les concepts de UML gardent leur notation.

UML-PP Méta-classe	UML Classe de base	Notation	Notation alternative
Step	Classifier	«Step» **Nom**	**Nom**
Step (qui invoque une tâche)	Classifier	«Step» **Nom** ⇨	**Nom** ⇨
Replicating Node	Classifier	•••	
Process Pattern	Classifier	(c.f.II.2) «ProcessPattern» **Nom**	**Nom**
ReuseFlow	Directed Relationship	« process pattern application » {parameter substitution} (c.f. II.3.1)	

Tableau II-5. Notations des méta-classes du modèle comportemental

La Figure II-32 montre un exemple de modèle comportemental.

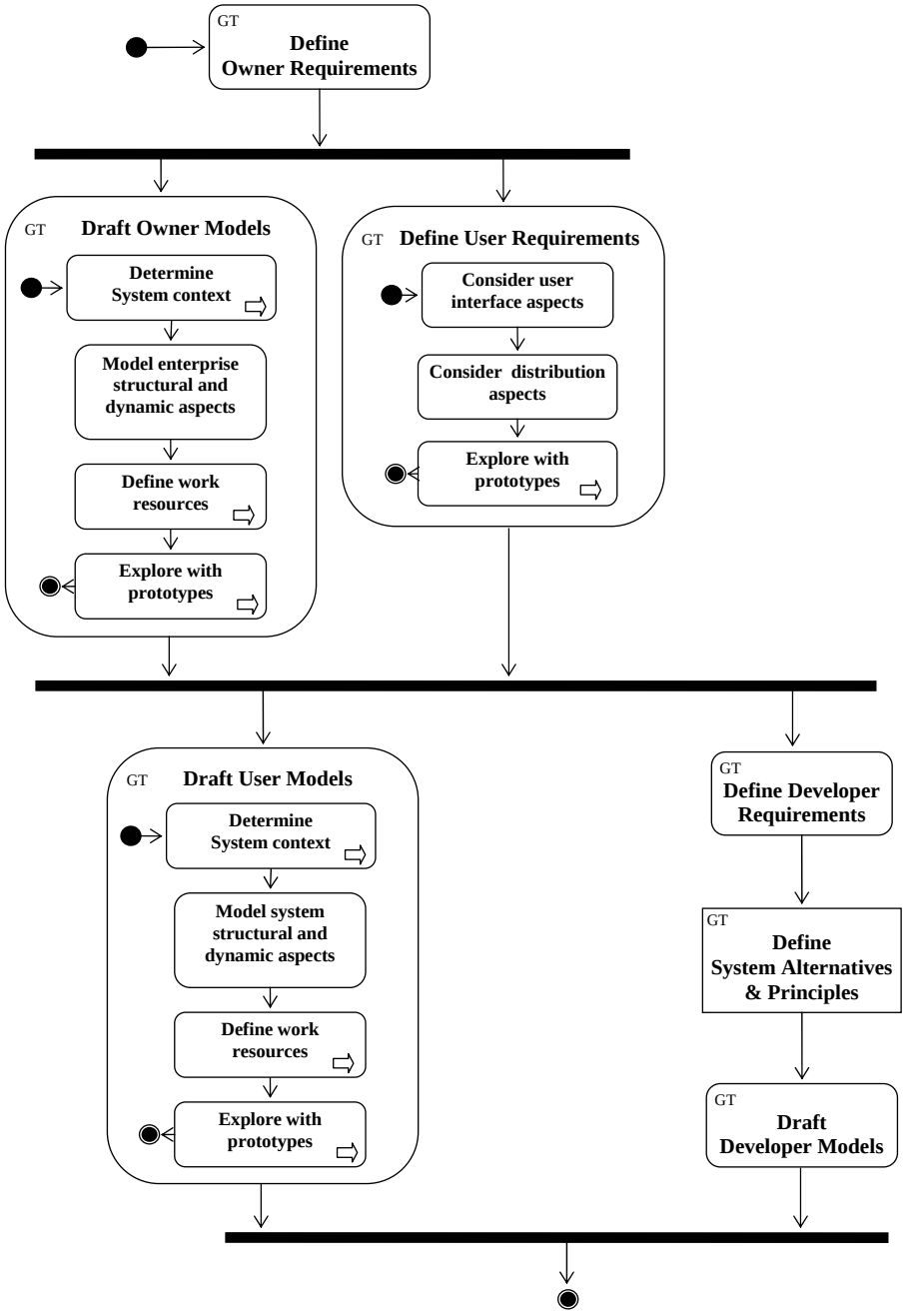


Figure II-32. Extrait de la phase *Preliminary Analysis* du procédé *Information System Delivery Process* de DMRMacroscope [SPEM05]

Dans cet exemple, le modèle comportemental est utilisé à deux niveaux : pour décrire l'enchaînement des tâches (*Define Owner Requirements, Draft Owner Models, Define User Requirements, Draft User Models*, etc.), et pour décrire le comportement d'une tâche (par exemple la tâche *Draft User Models*).

II.2. PAQUETAGE PROCESSPATTERN

Le paquetage *ProcessPattern* définit la structure interne d'un patron de procédé, et les relations pour appliquer et organiser les patrons de procédé.

Nous montrons dans la Figure II-33 l'extrait du méta-modèle représentant la structure d'un patron de procédé.

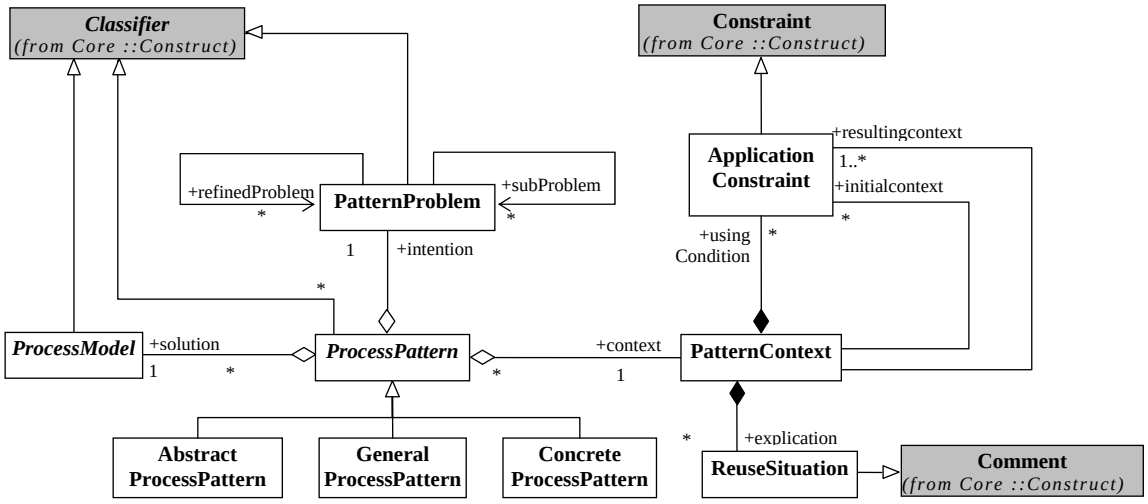


Figure II-33. Méta-modèle d'un patron de procédé

II.2.1. Patron de Procédé

Le concept de patron de procédé défini ici est décrit dans le méta-modèle UML-PP par la méta-classe *ProcessPattern* qui se compose d'un problème (*PatternProblem*), d'une solution représentée sous forme d'un (fragment de) modèle de procédé (*ProcessModel*), et d'un contexte spécifique (*PatternContext*) pour appliquer la solution du patron (Figure II-33).

Nous décrivons dans la suite les éléments constitutifs d'un patron de procédé.

II.2.2. Problème

Associé à un patron de procédé, un problème (*PatternProblem*) exprime l'intention du patron. Il est possible d'avoir plusieurs patrons qui résolvent le même problème.

Un problème peut se décomposer en plusieurs sous-problèmes (*subProblem*). Si une telle décomposition existe, cela signifie que pour traiter le problème supérieur, il faut résoudre tous les problèmes de niveau inférieur.

Un problème peut être raffiné. Le problème raffiné (*refinedProblem*) traite la même question que le problème initial mais sur des éléments ayant un niveau d'abstraction plus bas. Cela signifie que la solution du problème raffiné devra être plus détaillée.

Nous montrons dans la Figure II-34 des exemples de problèmes. Le problème *Control quality of a software artifact* est décomposé en deux sous-problèmes: *Verify an artifact* et *Rework an artifact*. Le problème *Verify an artifact* comporte à son tour deux sous-problèmes :

Find artifact's defects et *Evaluate artifact's defects*. Le problème *Verify an artifact* peut être raffiné en *Verify a RequirementDocument*, *Verify a DesignModel* ou *Verify a CodeModule*.

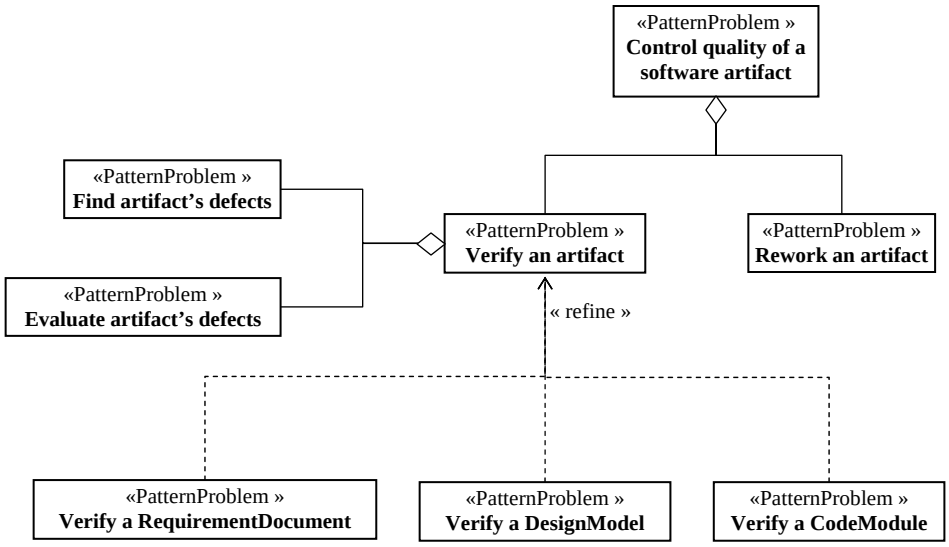


Figure II-34. Exemples de décomposition et de raffinement de problèmes

Dans la section I.2.1, nous avons catégorisé les problèmes de patrons de procédé en deux types : problème *de définition d'un élément de procédé* et problème *d'organisation d'éléments de procédé*.

Pour nous, le problème de définition d'un élément de procédé concerne tous les types d'éléments de procédé. Par exemple, la définition d'*une tâche de révision d'artéfact*, la définition d'un *rôle réalisant la tâche de révision*, et la définition *des rapports de révision* sont tous des problèmes de définition dans notre catégorisation.

Le problème d'organisation d'un ensemble d'éléments se porte en général sur les tâches mais peut concerner également des rôles et des produits. Par exemple le problème *modéliser des tâches collaboratives* concerne aussi la modélisation des produits manipulés par les tâches.

II.2.3. Contexte

Chaque patron de procédé a un contexte (*PatternContext*) d'application qui caractérise les conditions dans lesquelles le patron peut être appliqué et les résultats qu'il permet d'obtenir.

Le contexte du patron est composé de contraintes (*ApplicationConstraint*) qui décrivent les conditions nécessaires pour appliquer le patron *(initialcontext)* et les résultats attendus en appliquant ce patron *(resultingcontext)*, et d'une description de la réutilisation du patron *(ReuseSituation)*.

Dans la plupart des cas, les conditions d'application et les résultats sont exprimés par des contraintes sur les états des produits alors que le contexte de réutilisation décrit informellement d'autres contraintes concernant des ressources et l'environnement de développement (les participants, la limite de temps, l'infrastructure du projet, etc.).

Considérons par exemple le patron *FaganInspection* pour réviser statiquement un document en suivant une démarche bien définie basée sur des réunions et des discussions formelles. Le contexte initial de ce patron est que le document à réviser doit avoir été élaboré et que l'équipe de révision doit être prêt ; le contexte résultant est que l'artéfact révisé doit être approuvé, ainsi qu'un rapport de revue. La situation recommandée pour appliquer ce patron est que l'équipe soit bien organisée et centralisée, et que le document soit suffisamment complexe pour justifier la mise en œuvre de ce patron.

II.2.4. Solution

La solution d'un patron de procédé est représentée par un modèle de procédé Le niveau d'abstraction d'un tel modèle peut varier d'abstrait à concret selon le problème adressé par le patron (c.f. II.1.6).

II.2.5. Typologie de Patrons de Procédé

Comme nous l'avons expliqué dans la section I.3, nous distinguons plusieurs types de patrons de procédé selon leurs *niveaux d'abstraction*.

AbstractProcessPattern

Un patron abstrait capture un modèle de procédé qui fournit une structure récurrente et générique pour modéliser ou organiser des éléments de procédé (c.f la Figure II-35 pour un exemple de patron abstrait). La solution d'un patron abstrait contient au moins un élément abstrait[1]. En général, ce type de patron traite des problèmes d'organisation.

GeneralProcessPattern

Un patron général capture un modèle de procédé qui ne contient aucun élément abstrait, mais comprend des éléments généraux qui ont des sémantiques générales, sans être complètement spécifiés (c.f. la Figure II-36 pour un exemple de patron général). Un patron général peut s'appliquer pour définir le contenu d'un élément général, ou pour organiser un groupe d'éléments généraux.

ConcreteProcessPattern

Un patron concret capture un modèle de procédé ayant une sémantique précise. Autrement dit, il ne contient que des éléments concrets qui sont complètement spécifiés (c.f. la Figure II-37 pour un exemple de patron concret). Un patron concret peut s'appliquer pour définir le contenu d'un élément concret, ou pour organiser un groupe d'éléments concrets.

Opérations additionnelles

Nous définissons une méta-opération pour vérifier la conformité du niveau d'abstraction entre deux patrons de procédé :

[1] Autrement dit, un patron abstrait peut aussi contenir des éléments généraux ou concrets.

[OP8] La méta-opération *isConformTo()* retourne la valeur True si le niveau d'abstraction du patron passé en paramètre est conforme à celui du patron à comparer.

```
ProcessPattern::isConformTo (p: ProcessPattern): Boolean;
if self.oclIsTypeOf(AbtractProcessPattern)
   isConformTo = p.oclIsTypeOf(AbstractProcessPattern)
                     or p.oclIsTypeOf(GeneralProcessPattern)
                     or p.oclIsTypeOf(ConcreteProcessPattern)
else
    if self.oclIsTypeOf(GeneralProcessPattern)
    isConformTo = p.oclIsTypeOf(GeneralProcessPattern)
                     or p.oclIsTypeOf(ConcreteProcessPattern
else  if self.oclIsTypeOf(ConcretProcessPattern)
   isConformTo = p.oclIsTypeOf(ConcreteProcessPattern
```

Règles de bonne modélisation

Les contraintes suivantes sont définies sur les constituants de chaque type de patron:

[C33] La solution d'un patron abstrait doit contenir au moins un élément abstrait.

```
context ProcessPattern inv:
self.oclIsTypeOf(AbstractProcessPattern)implies
self.solution.element → select(e|e.oclIsKindOf(ProcessElement))→
                              exists(e|e.getAbstractionLevel='Abstract')
```

[C34] La solution d'un patron général n'a aucun élément abstrait, et doit contenir au moins un élément général.

```
context ProcessPattern inv:
self.oclIsTypeOf(GeneralProcessPattern)implies
self.solution.element → select(e|e.oclIsKindOf(ProcessElement))→
         excludes(e|e.getAbstractionLevel='Abstract')
                      and exists(e|e.getAbstractionLevel='General')
```

[C35] La solution d'un patron concret ne contient que des éléments concrets.

```
context ProcessPattern inv:
self.oclIsTypeOf(ConcreteProcessPattern)implies
self.solution.element → select(e|e.oclIsKindOf(ProcessElement))→
                               forAll(e|e.getAbstractionLevel='Concrete')
```

Notation

Un patron est en fait un sorte de paquetage qui encapsule différents éléments pour décrire une solution d'un problème et son contexte d'application. Nous adaptons donc la notation de *Package* pour représenter des patrons de procédé. Pour des raisons de simplicité, le problème, le contexte et la solution d'un patron sont représentés comme des éléments internes au patron.

Les notations relatives aux patrons de procédé sont représentées dans le Tableau II-6.

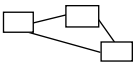
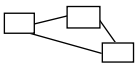
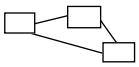
UML-PP Méta-classe	**UML Classe de base**	**Notation**	**Notation alternative**
Abstract ProcessPattern	Classifier	«AbtractProcessPattern» **Nom** Problem Initial context Resulting context Reuse Situation Solution 	APP **Nom**
General ProcessPattern	Classifier	«GeneralProcessPattern» **Nom** Problem Initial context Resulting context Reuse Situation Solution 	GPP **Nom**
Concrete ProcessPattern	Classifier	«ConcreteProcessPattern» **Nom** Problem Initial context Resulting context Reuse Situation Solution 	CPP **Nom**

Tableau II-6. Notations pour les patrons de procédé

Nous donnons dans la suite des exemples illustrant les trois types de patron de procédé.

La Figure II-35 montre le patron *ClonableLifeCycle* extrait du *Microsoft Solution Framework* par Pavlov et Malenko [Pavlov04].

Ce patron vise un problème connu du développement concernant la situation dans laquelle certaines informations arrivent après que le processus ait déjà commencé (par exemple, un nouveau risque peut être découvert au milieu du processus de gestion des risques). De telles informations tardives peuvent remettre en cause des décisions prises antérieurement dans le processus. Par conséquent, elles ne peuvent pas être ignorées et devraient être prises en considération dans toutes les étapes de processus. Le patron *ClonableLifeCycle* adresse donc un problème d'organisation d'éléments de procédé. Sa solution capture une structure de procédé générique permettant de prendre en compte des informations tardives. Dans la structure proposée, une tâche peut retourner au début du processus lorsqu'une nouvelle information apparaît. Ce patron étant applicable à tout procédé, nous le décrivons comme un patron abstrait.

«AbstractProcessPattern»
Cloneable LifeCycle

Problem : Cope with a new artefact that appears during the last steps of a process and can influence previous decisions.

Initial context: *NewArtefact* is available when the first steps of process needed to proceed it are executed

Resulting context: *NewArtefact* is gone through all process steps.

Reuse Situation: when there is a need to incorporate efficiently a new artefact appearing after certain steps of a process.

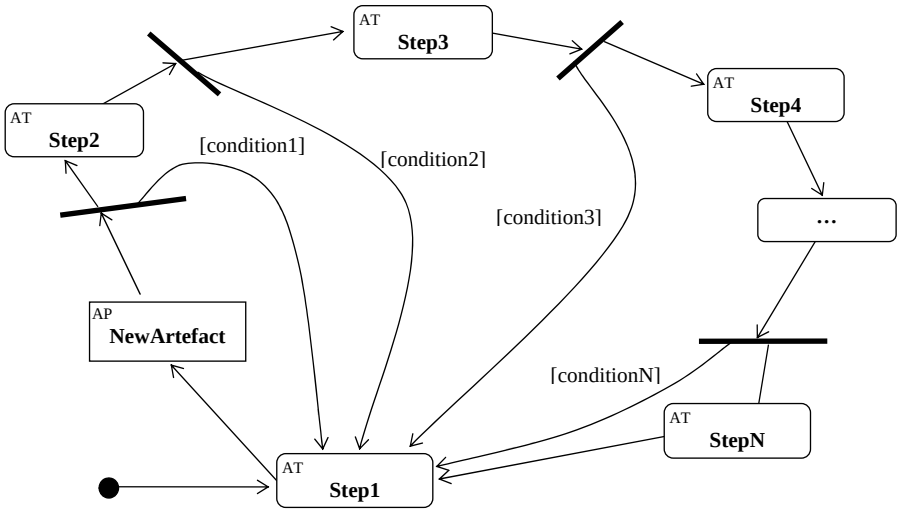


Figure II-35. Un patron de procédé abstrait

La Figure II-36 montre le patron *FaganInspection* [Fagan86] qui propose une démarche pour vérifier un artéfact de développement.

Ce patron vise donc un problème de définition de tâche. La solution du patron est applicable pour vérifier plusieurs types d'artéfacts (par exemple des besoins, des modèles de conception ou des codes). Elle est donc décrite avec des éléments généraux et capturée dans un patron général.

La Figure II-37 montre le patron *MyRUPSoftwareArchitectRole* qui propose une définition concrète pour le rôle *SoftwareArchitect* adapté de celui de RUP [Kruchten03] Ce rôle est défini en principe pour une petite équipe qui adopte l'approche RUP et le langage UML.

«GeneralProcessPattern»
FaganInspectionPattern

Problem : Detect Product defects

Initial context: *Artifact* is complete for review; inspection roles, resources (standards, tools) are available.

Resulting context: *Artifact* is approved

Reuse Situation: when one needs to verify statically the correctness of an artifact by using a formal group meeting based process and to get feedback for improving process itself.

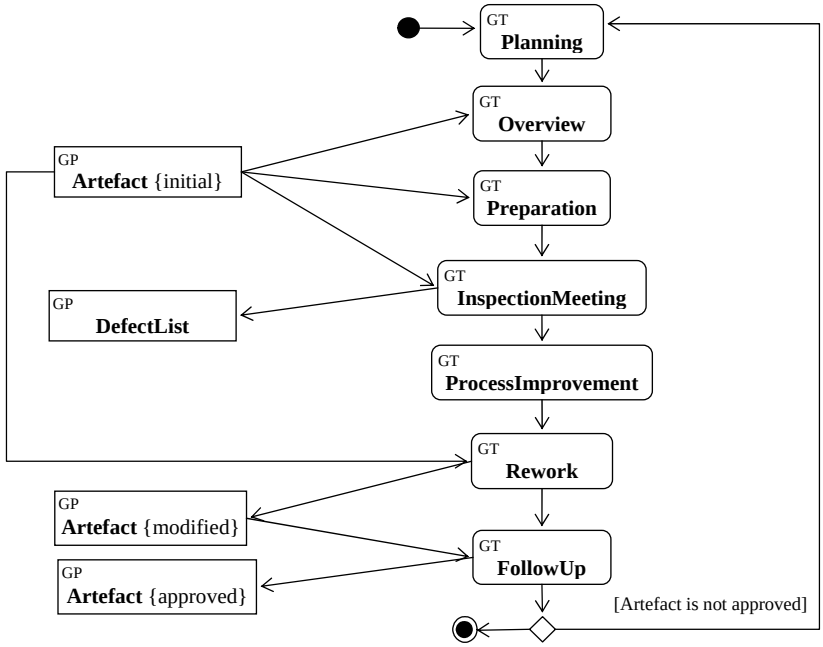


Figure II-36. Un patron de procédé général

«ConcreteProcessPattern»
MyRUPSoftwareArchitectRole

Problem : Definition of Software Architect role

Initial context: a RUP–based process is adopted, *SoftwareArchitect Role* is not defined yet

Resulting context: *SoftwareArchitect Role* is described in details.

Reuse Situation: when one needs to define the responsibilities and the skills of the *SoftwareArchitect Role* in an RUP-adapted process for small teams and simple projects.

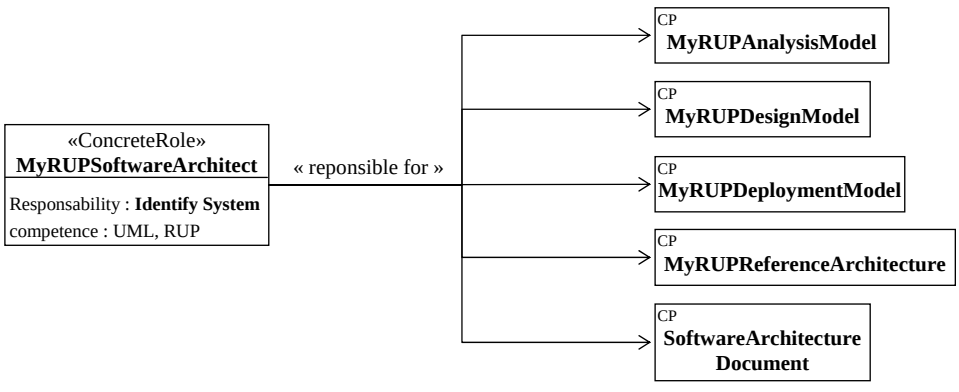


Figure II-37. Un patron de procédé concret

II.3. PAQUETAGE PATTERNRELATIONSHIP

Dans ce paquetage, nous définissons les relations d'application et les relations d'organisation de patrons de procédé. Les relations d'application sont représentées sous forme de relations établies entre éléments de procédé et patrons. Elles permettent de décrire des éléments de procédé en réutilisant des patrons de procédé. Les relations d'organisation expriment les liens entre patrons de procédé. Elles facilitent la sélection et l'application de patrons.

La Figure II-38 montre diverses relations entre patrons et éléments de procédé définis dans notre méta-modèle.

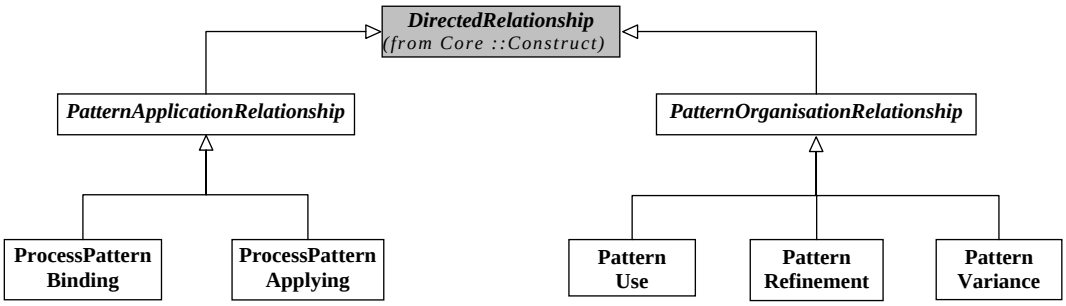


Figure II-38. Relations entre patrons et éléments de procédé

Dans la suite, nous présentons d'abord les relations d'application de patrons, puis les relations d'organisation de patrons.

II.3.1. Relations d'application de Patrons de Procédé

Nous présentons dans cette section les concepts de UML-PP permettant la représentation de la réutilisation de patrons de procédé dans des modèles de procédé. Cette réutilisation est exprimée aux moyens de deux relations dont l'intérêt a été illustré informellement dans la section I.4.

Comme nous l'avons montré dans la section II.2.5, les patrons de procédé peuvent exister à différents niveaux d'abstraction. Pour promouvoir l'utilisation des patrons, il faut permettre de les réutiliser à n'importe quel niveau d'abstraction ; autrement dit, il faut employer le mécanisme d'abstraction pour réutiliser les patrons de procédé. Dans cette intention, nous définissons les patrons de procédé comme des éléments paramétrés (c.f. II.3.1.1).

Pour exprimer l'application de patrons de procédé dans des modèles de procédé, nous proposons les relations *ProcessPatternBinding* (c.f. II.3.1.2) et *ProcessPatternApplying* (c.f. II.3.1.3) qui supportent respectivement *la définition d'un élément* de procédé et *l'organisation d'un groupe* d'éléments de procédé, selon les besoins de réutilisation de patrons de procédé identifiées dans la section I.4.

II.3.1.1. Paramétrisation de patrons de procédé

La Figure II-39 montre l'extrait du méta-modèle UML-PP définissant un patron de procédé comme un élément paramétré.

Un *ProcessPattern* peut avoir une *ProcessPatternSignature* qui spécifie un ensemble de paramètres formels pour le patron. Un *ProcessPatternParameter* référence un *ProcessElement* ou un *ProcessPattern* qui peut être substitué dans une relation d'application. L'attribut «*optional*» d'un paramètre permet de spécifier s'il est indispensable (*optional* = False) ou non. Un patron de procédé contenant une signature est donc un patron paramétré désigné aussi par le terme de *template*.

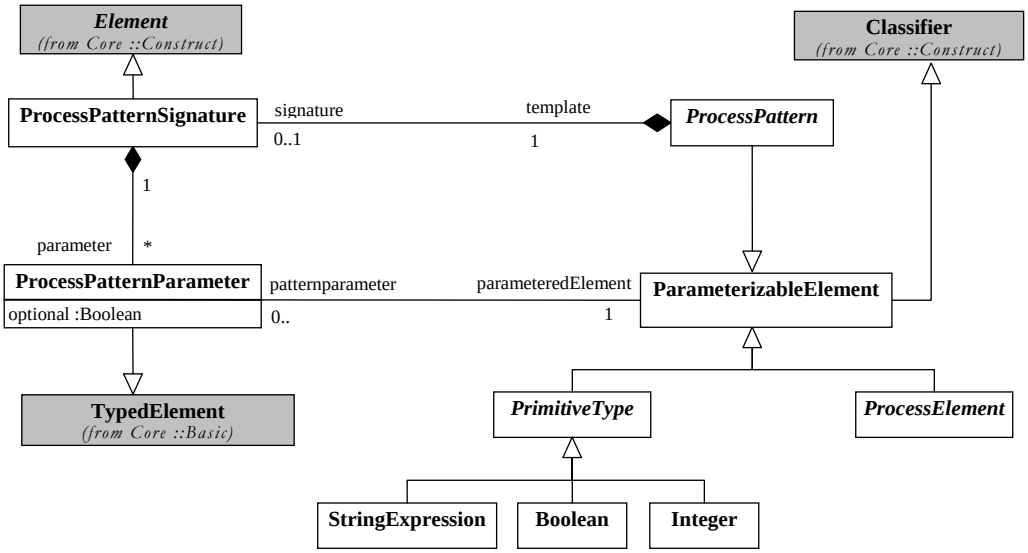


Figure II-39. Méta-modèle d'un patron paramétré

Règles de bonne modélisation

Pour garantir une représentation cohérente et logique des patrons de procédé, nous définissons les contraintes suivantes :

[C36] Seul les patrons abstraits ou généraux peuvent être paramétrés.

```
context ProcessPattern inv:
self.signature → notEmty() implies
self.oclIsTypeOf(AbstractProcessPattern)
      or self.oclIsTypeOf(GeneralProcessPattern)
```

[C37] Les éléments représentant les paramètres formels d'un patron doivent être des éléments de procédé ou des éléments de types primitifs (*integer*, *boolean* ou *string*).

```
context ProcessPattern inv:
let formalparameters = self.signature.parameter.parameteredElement
formalparameters → forAll(e|e.oclIsTypeOf(ProcessElement)or
     e.oclIsTypeOf(Integer)or e.oclIsTypeOf(Boolean or e.oclIsTypeOf(String))
```

[C38] Les éléments représentant les paramètres formels d'un patron doivent être inclus dans le modèle représentant la solution du patron.

```
context ProcessPattern inv:
let formalparameters = self.signature.parameter.parameteredElement and
let solutionelements = self.solution.element
self.signature → notEmty() implies solutionelements → formalparameters
```

Notation

Le Tableau II-7 montre la notation adoptée pour un patron paramétré.

UML-PP Méta-classe	UML Classe de base	Notation	Notation alternative
ProcessPattern	Classifier	«*ProcessPattern*» **Nom** **Signature** Problem Initial context Resulting context Reuse Situation Solution	*PP* **Nom**

Tableau II-7. Notations pour les patrons de procédé paramétrés

II.3.1.2. Relation ProcessPatternBinding

Afin d'exprimer l'utilisation d'un patron de procédé pour générer le contenu d'un élément de procédé, nous définissons la relation *ProcessPatternBinding*. Cette relation est définie entre un élément de procédé (source) et un patron de procédé (cible) (Figure II-40).

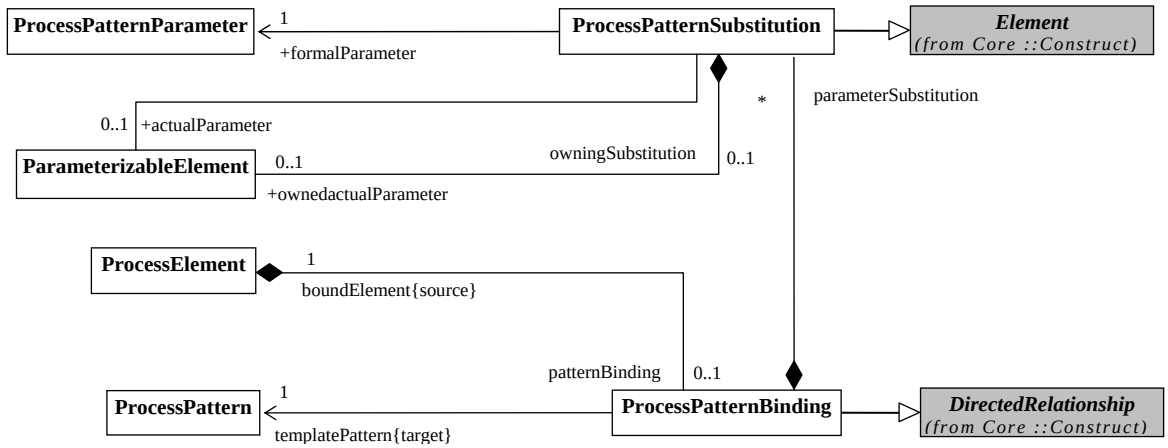


Figure II-40. Méta-modèle de la relation *ProcessPatternBinding*

Une relation *binding* exprime le lien de trace existant entre un élément de procédé et un template utilisé pour sa conception. La présence d'une relation *ProcessPatternBinding* implique que le modèle de procédé associé au patron cible (*templatePattern*) sera copié dans l'élément source (*boundElement*) en substituant via *ProcessPatternsSubstitution* les paramètres formels du patron (*formalParameter*) par les éléments correspondants passés en paramètres effectifs (*actualParameter*). Il n'est pas nécessaire que les paramètres effectifs soient des éléments existants, c'est-à-dire des éléments créés dans le modèle contenant l'élément en cours de définition. Pour permettre la génération d'éléments nouveaux, une relation *ProcessPatternsBinding* peut accepter un paramètre effectif inexistant, déclaré sous forme de *StringExpression* (c.f. [C42][C43]).

Si un paramètre effectif est un élément existant, il faut vérifier sa conformité (copatibilité) avec le paramètre formel correspondant. Par exemple, si le type d'un paramètre formel est *GeneralTask*, celui du paramètre effectif doit être *GeneralTask* ou *ConcretTask* mais pas un produit ou un rôle.

Pour pouvoir être réutilisé pour définir un élément, un patron doit couvrir les besoins de l'élément. Autrement dit, un patron template plus général peut être spécialisé pour satisfaire la spécification d'un élément plus spécifique, mais pas l'inverse[1].

Règles de bonne modélisation

Les contraintes suivantes sont définies pour garantir la cohérence de l'utilisation de la relation *ProcessPatternBinding* :

[C39] Le niveau d'abstraction de l'élément attaché (*boundElement*) doit être inférieur ou égal à celui du patron cible.

```
context ProcessPatternBinding inv:
let boundAbs = boundElement.getAbstractionLevel()
templatePattern.oclIsTypeOf(AbstractProcessPattern)implies
 boundAbs="Abstract" or boundAbs="General" or boundAbs="Concrete"
templatePattern.oclIsTypeOf(GeneralProcessPattern)implies
 boundAbs="General" or boundAbs="Concrete"
templatePattern.oclIsTypeOf(ConcreteProcessPattern)implies
 boundAbs="Concrete"
```

[C40] Chaque substitution de paramètre doit référencer un paramètre formel du patron cible.

```
context ProcessPatternBinding inv:
parameterSubstitution->forAll(b| templatePattern.signature.parameter
                                 → includes(b.formalParameter))
```

[C41] Une relation *ProcessPatternBinding* contient au plus une substitution de paramètre pour chaque paramètre formel du patron cible.

```
context ProcessPatternBinding inv:
templatePattern.signature.parameter → forAll(p| parameterSubstitution
                    →select(b | b.formalParameter = p)->size() <= 1)
```

[C42] Si un paramètre effectif est un élément existant, il doit être compatible avec le paramètre formel correspondant, c'est-à-dire que le type du paramètre effectif doit être le même ou un sous-type de celui du paramètre formel.

```
context ProcessPatternBinding inv:
self.parameterSubstitution.actualParameter→
    select(a|boundElement.owningModel→exists(a))→
          forAll(e|e.isCompatibleWith(formalParameter.parameteredElement))
```

[1] Par exemple, on ne peut pas utiliser un patron concret pour générer le contenu d'un élément général. En effet, l'élément à définir perdrait son caractère général.

[C43] Si un paramètre effectif est déclaré avec le type StringExpression, cela signifie qu'il n'existe pas dans le modèle comportant l'élément attaché (*boundElement*).

```
context ProcessPatternBinding inv:
self.parameterSubstitution.actualParameter
          →select(a|a.oclIstypeOf(StringExpression))
                    →forAll(e| not boundElement.owningModel→includes(e))
```

Notation

Le Tableau II-8 montre la notation représentant la relation *ProcessPatternBinding*.

UML-PP Méta-classe	UML Classe de base	Notation
ProcessPattern Binding	Directed Relationship	**Process Element** — « bound to » < {formal parameter ᵢ→actual parameter ᵢ } > → **Process Pattern**

Tableau II-8. Notation de la relation *ProcessPatternBinding*

Dans la Figure II-41 ci-dessous nous développons l'exemple du début de la section II.3.1 (c.f. Figure II-7) afin d'illustrer l'utilisation de la relation *ProcessPatternBinding* pour représenter la génération des contenus des éléments de procédé. Les résultats de ces applications de patrons sont montrés dans la Figure II-42.

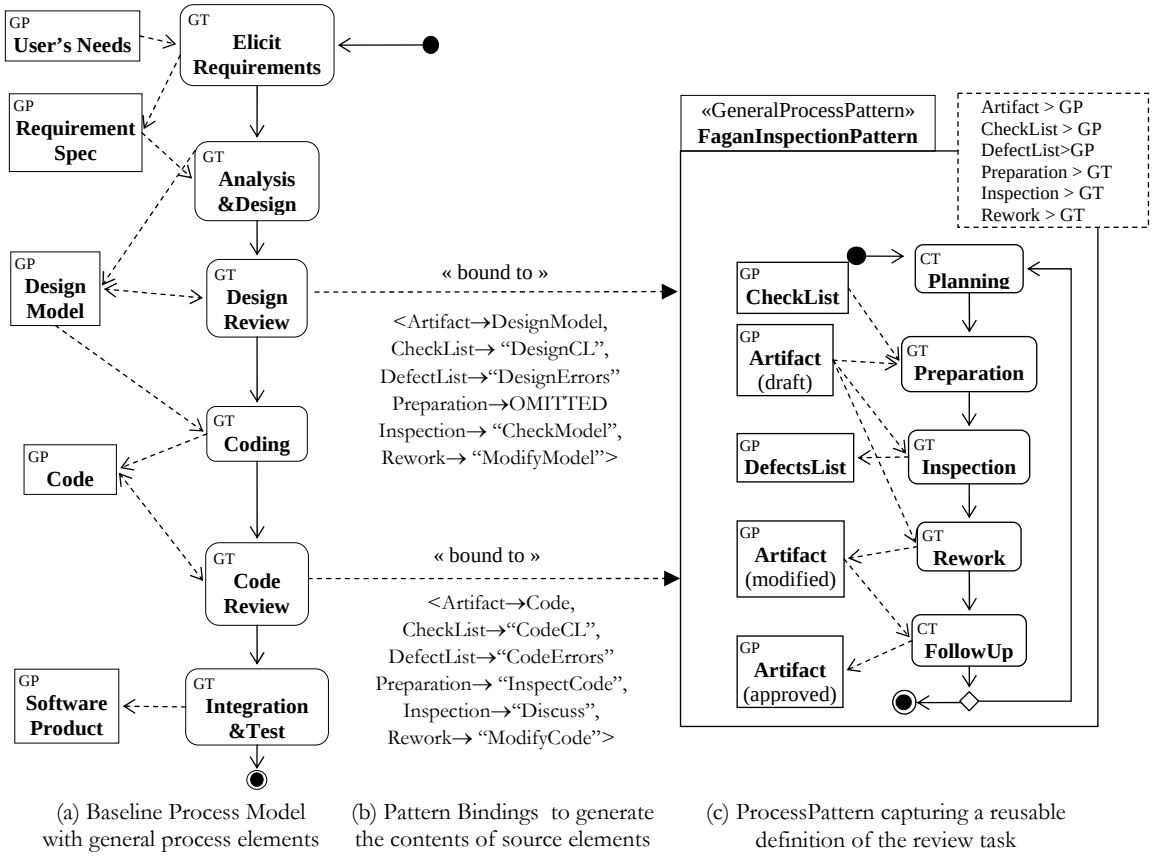


Figure II-41. Exemples d'applications de patrons pour générer des éléments de procédé

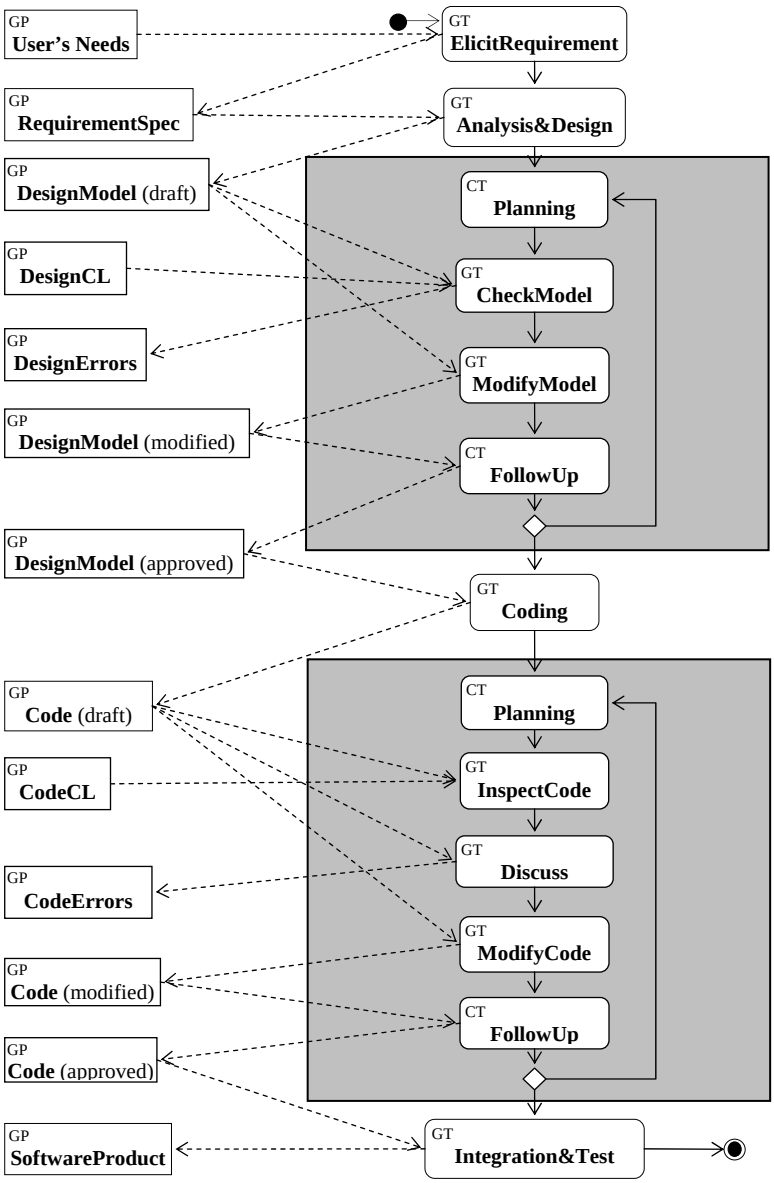


Figure II-42. Résultats du dépliage de la relation *ProcessPatternBinding* sur le modèle de procédé de la Figure II-41

La Figure II-41a montre un modèle de procédé de développement contenant des éléments de procédé généraux, i.e. non complètement spécifiés. On suppose que dans la base de patrons de procédé de l'entreprise, le concepteur de procédé a trouvé le patron *FaganInspectionPattern* (Figure II-41c) vu précédemment. Ce patron général connu peut être appliqué pour réviser différents types de produits (conception, code, etc.). Il est défini avec six paramètres : trois produits *(Artifact CheckList* et *DefectList)*, et trois tâches (*Preparation, Inspection* and *Rework*). Les types de ces paramètres formels étant conformes à des éléments généraux, ils peuvent être substitués par des paramètres effectifs ayant un niveau d'abstraction inférieur ou égal, c'est-à-dire de niveau général ou concret. Dans ce patron, il y a également deux tâches concrètes (*Planification* et *FollowUp*) qui restent les mêmes pour n'importe quelle tâche de révision. Par conséquent, elles ne sont pas traitées comme paramètres.

La Figure II-41b illustre l'utilisation de la relation *ProcessPatternBinding* pour produire les contenus des tâches générales *DesignReview* et *CodeReview* à partir du patron *FaganInspectionPattern*. Dans l'application de *binding* sur la tâche *DesignReview*, seul le paramètre effectif *DesignModel* est un élément existant, les autres étant déclarés sous forme de *StringExpression*. La substitution du paramètre formel *Préparation* par la valeur OMITTED signifie que l'élément en paramètre formel sera omis dans le modèle résultant. Dans l'application de *binding* sur la tâche *CodeReview*, seul le paramètre effectif *Code* est un élément existant.

II.3.1.3. Relation ProcessPatternApplying

Afin de permettre l'utilisation d'un patron pour (re)organiser ou enrichir un ensemble d'éléments de procédé, nous proposons la relation *ProcessPatternApplying*. Cette relation est définie entre un patron de procédé (source) et un modèle de procédé (cible) (Figure II-43).

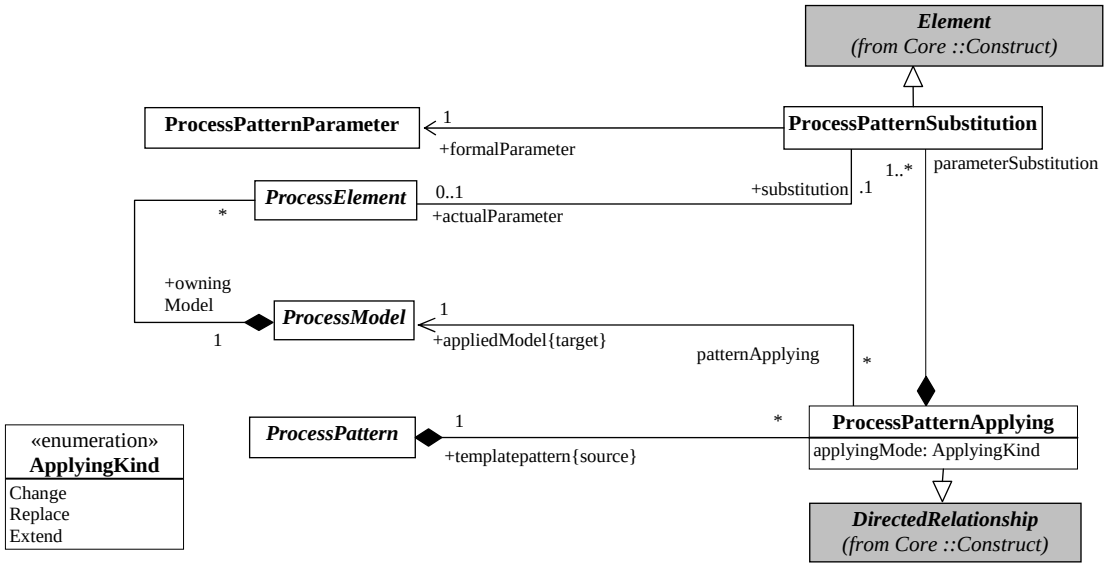


Figure II-43. Méta-modèle de la relation *ProcessPatternApplying*

La structure de la méta-classe *ProcessPatternApplying* est comparable à celle de la méta-classe *ProcessPatternBinding* présentée ci-dessus. Cependant, sa sémantique est différente. Une relation *ProcessPatternApplying* lie un patron source (*templatePattern*) à un modèle de procédé cible (*appliedModel*) en mettant en relation chaque élément du patron défini comme paramètre formel avec un élément existant dans le modèle cible défini comme paramètre effectif.

La manière dont cette application est réalisée dépend du mode d'application (attribut *applyingMode*) choisi. Pour couvrir les principaux besoins de modélisation, nous avons défini trois valeurs possibles pour cet attribut :

- ***applyingMode* = Change**

Si le mode *«Change»* est choisi, la solution du patron source remplace le contenu du modèle cible. Plus précisément, les contenus des éléments du modèle cible représentant des paramètres effectifs sont remplacés par ceux des paramètres formels du patron. Cela peut conduire à un changement de niveau d'abstraction des éléments représentant des paramètres effectifs. Les relations (si elles existent) entre les éléments du modèle cible représentant des

paramètres effectifs sont aussi remplacées par celles définies entre les paramètres formels correspondants dans le patron source.

Ce mode d'application est choisi quand on veut appliquer un patron pour enrichir et éventuellement réorganiser un modèle de procédé. Dans ce cas, le niveau d'abstraction du patron source doit être inférieur ou égal à celui du modèle cible (c.f. [C44]). Par exemple on peut réutiliser un patron contenant des éléments concrets pour raffiner le contenu d'un modèle composé d'éléments généraux, mais on ne peut pas utiliser un patron abstrait pour ce but.

- ***applyingMode* = Replace**

Le mode « *Replace*» indique que le contenu du patron source sera fusionné avec celui du modèle cible en favorisant le patron source. Plus précisément, dans le cas où le modèle cible contient des éléments et des relations différents de ceux du patron source, ces éléments existants seront conservés dans le modèle résultant s'ils ne sont pas en contradiction avec ceux définis dans le patron source. Sinon ils seront remplacés par les éléments du patron source.

- ***applyingMode* = Extend**

Le mode «*Extend*» indique que le contenu du patron source sera fusionné avec celui du modèle cible en favorisant le modèle cible. Dans ce cas, les éléments existants du modèle cible restent intacts et les nouveaux éléments du patron sources sont ignorés s'il y a des conflits entre eux.

Les modes d'application *«Replace»* et *«Extend»* sont choisis quand on veut appliquer un patron pour réorganiser un modèle de procédé en conservant les éléments du modèle cible représentant des paramètres effectifs. Plus précisément, après la fusion, le contenu et le niveau d'abstraction des éléments du modèle cible ne changent pas mais les relations entre eux peuvent changer. Dans ces deux modes, le niveau d'abstraction du patron source doit être égal ou supérieur à celui du modèle cible pour fournir une solution conforme à, ou plus générique que celle du modèle cible (c.f.[C45]).

En établissant une relation *ProcessPatternApplying*, le concepteur doit mettre en relation des éléments du patron source avec des éléments existant dans le modèle cible par spécification des substitutions de paramètres. Pour garantir une bonne correspondance entre les paramètres formels et les paramètres effectifs, il faut vérifier qu'ils sont compatibles (c.f.[C50]).

Des conflits peuvent naturellement se produire dans plusieurs situations. Ils peuvent résulter d'incohérences sur des dépendances dans l'ordre d'exécution, sur des impacts entre des produits, sur des relations entre des rôles et des tâches, etc. Les types de conflits potentiels liés à cette relation *ProcessPatternApplying* sont discutés dans la section III.2 du Chapitre III.

ProcessPatternApplying est une relation établie entre un patron et des éléments existants. Par conséquent, en spécifiant cette relation, le concepteur doit spécifier explicitement la substitution des paramètres pour permettre une application correcte du patron.

Règles de bonne modélisation

Les contraintes suivantes sont définies pour garantir la cohérence de l'utilisation de la relation *ProcessPatternApplying* :

[C44] Si le mode d'application est *«Change»*, le niveau d'abstraction du modèle cible doit être supérieur ou égal à celui du patron source.

```
context ProcessPatternApplying inv:
self.applyingMode="Change" implies
    templatePattern.isCompatibleWith(appliedModel)
```

[C45] Si le mode d'application est *«Replace»* ou *«Extend»,* le niveau d'abstraction du modèle cible doit être inférieur ou égal à celui du patron source.

```
context ProcessPatternApplying inv:
(self.applyingMode="Replace")or (self.applyingMode="Extend") implies
      appliedModel.isCompatibleWith(templatePattern)
```

[C46] Une relation *ProcessPatternApplying* doit avoir au moins une substitution de paramètre.

```
context ProcessPatternApplying inv:
parameterSubstitution->size() >= 1
```

[C47] Chaque substitution de paramètre doit référencer un paramètre formel du patron source.

```
context ProcessPatternApplying inv:
parameterSubstitution->forAll(b| templatePattern.signature.parameter
                                  → includes(b.formalParameter))
```

[C48] Une relation *applying* contient au plus une substitution de paramètre pour chaque paramètre formel du patron cible.

```
context ProcessPatternApplying inv:
templatePattern.signature.parameter → forAll(p| parameterSubstitution
                     →select(b | b.formalParameter = p)->size() <= 1)
```

[C49] Les éléments utilisés comme paramètres effectifs doivent appartenir au modèle cible.

```
context ProcessPatternApplying inv:
self.parameterSubstitution.actual->forAll (p|
                    self.appliedModel.allOwnedElement()->includes (p))
```

[C50] Un paramètre effectif doit être compatible avec le paramètre formel correspondant, c'est-à-dire que le type du paramètre effectif doit être le même ou un sous-type de celui du paramètre formel.

```
context ProcessPatternApplying inv:
self.parameterSubstitution.actualParameter
    →select(a|boundElement.owningModel→exists(a))
         → forAll(e| e.isCompatibleWith(formalParameter.parameteredElement))
```

Notation

Le Tableau II-9 montre la notation représentant la relation *ProcessPatternApplying*.

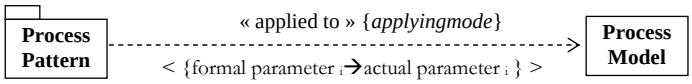
UML-PP Méta-classe	UML Classe de base	Notation
ProcessPattern Applying	Directed Relationship	 {*applyingmode*} spécifie la valeur de l'attribut *applyingMode* de la relation *ProcessPatternApplying*.

Tableau II-9. Notation de la relation *ProcessPatternApplying*

Nous présentons dans la Figure II-44 l'utilisation de la relation *ProcessPatternApplying* pour réorganiser la structure du modèle de procédé de la Figure II-8.

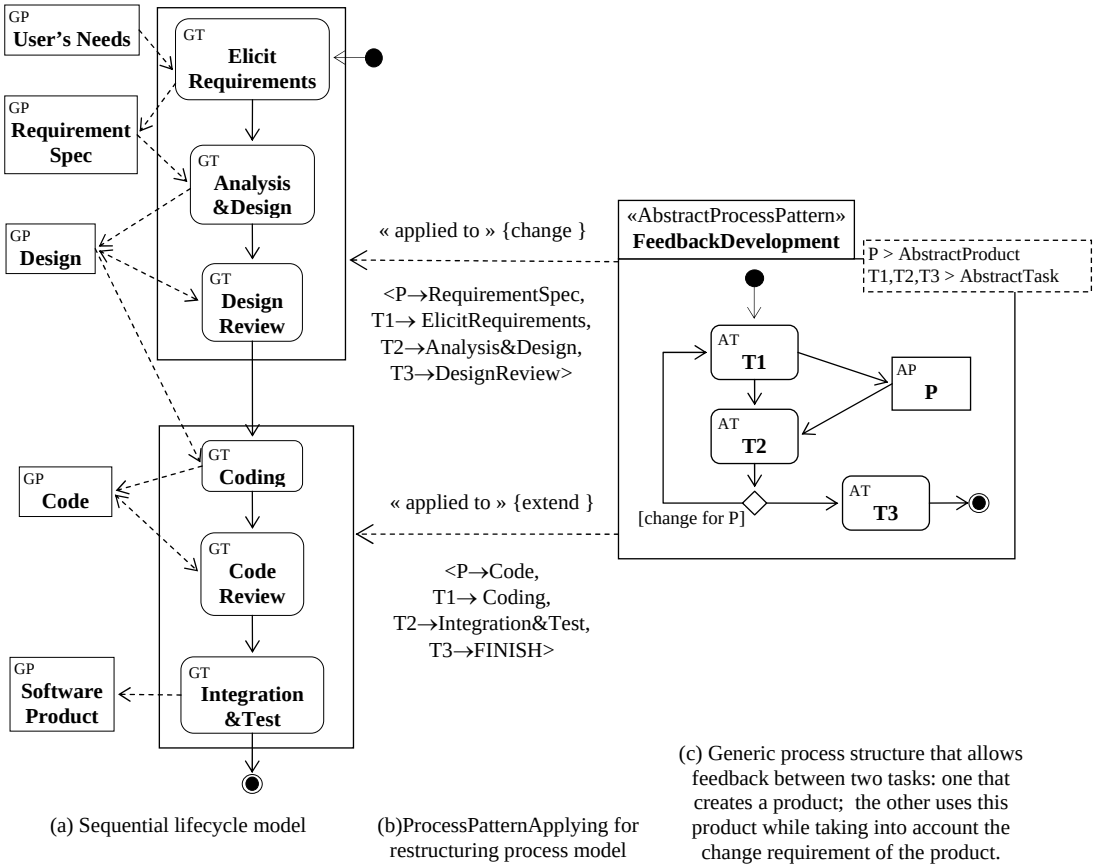


Figure II-44. Exemples d'applications de patrons pour organiser des éléments de procédé

Le cycle de vie séquentiel du procédé de la Figure II-44a ne permet pas de traiter des changements durant l'exécution de procédé. Le concepteur veut donc adopter un autre modèle de cycle de vie supportant des rétroactions. Cependant, il souhaite garder les éléments originels du procédé qui ont été spécifiés. Pour ce faire, il peut employer la relation *ProcessPatternApplying* pour réorganiser les éléments existants.

Supposons que le concepteur ait choisi le patron *FeedbackDevelopment* (Figure II-44c) qui capture une structure de procédé générique représentant un ordre d'exécution avec des rétroactions entre deux tâches. Ce patron adresse le problème classique de changement des produits élaborés par des étapes antérieures dans un processus. Il est donc défini comme un patron abstrait ayant quatre paramètres formels : un produit abstrait *P*, et trois tâches abstraites *T1*, *T2*, *T3*. *T1* produit *P* et alimente *T2* qui consomme *P*. A la fin de l'exécution de *T2*, le besoin de changement de *P* est considéré. Si *P* doit être changé alors *T1* est de nouveau exécutée, sinon *T3* est exécutée.

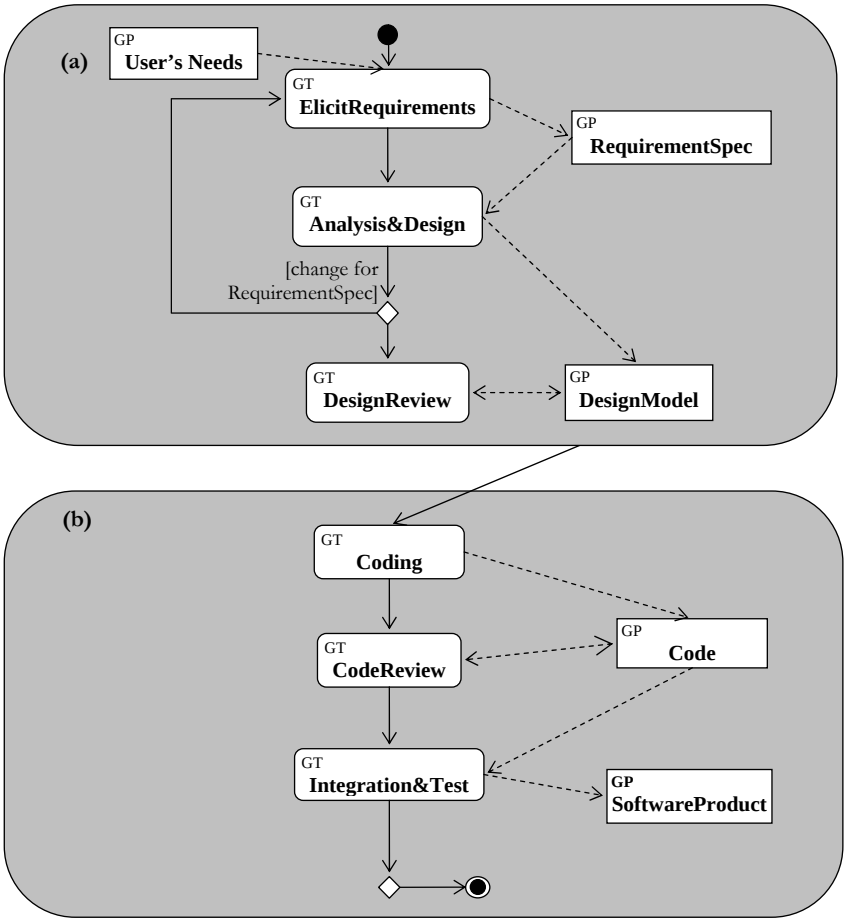


Figure II-45. Résultat des dépliages de la relation *ProcessPatternApplying* sur le modèle de procédé de la Figure II-44

Le patron choisi est utilisé deux fois pour restructurer le procédé originel. D'abord, le patron *FeedbackDevelopment* est appliqué au modèle de procédé contenant les éléments de procédé suivants: *RequirementSpec, ElicitRequirements, Analysis&Design* et *DesignReview* (Figure II-44b). Ces éléments jouent respectivement les rôles de *P*, *T1*, *T2*, et *T3* dans le patron source. Cette première application est spécifiée avec le mode *Change*. Par conséquent, les relations entre *RequirementSpec, ElicitRequirements, Analysis&Design* et *DesignReview* sont supprimées et remplacées par celles du patron *FeedbackDevelopment*. Après l'application, on constate que les relations entre *ElicitRequirements, Analysis&Design* et *DesignReview* ont été rétablies et que la rétroaction entre *T2* et *T1* a été ajoutée au modèle cible (Figure II-45a).

En revanche, la deuxième application du modèle *FeedbackDevelopment* au modèle de procédé contenant *Code*, *Coding*, *CodeReview*, *Integration&Test* et FINISH[1], est une fusion réalisée avec le mode *Extend*. Dans cette application, *Code*, *Coding*, *Integration&Test* et FINISH remplacent respectivement *P, T1, T2* et *T3* dans le modèle. Cependant, le modèle cible contient un autre élément, la tâche *CodeReview*. Cet élément additionnel est conservé dans le modèle résultant parce que le nouveau flux de contrôle entre *Coding* et *Integration&Test* n'est pas en conflit avec les relations existant entre *CodeReview* et les autres éléments (Figure II-45b).

II.3.2. Relations d'organisation de Patrons de Procédé

Si un patron est une solution récurrente à un problème dans un contexte donné, un système de patrons est une collection de solutions qui coopèrent pour résoudre un problème complexe. Organisés en de tels systèmes[2], les patrons peuvent exprimer pleinement leur force .

Pour représenter un système de patrons, il faut définir les relations entre patrons. De telles relations sont appelées relations d'organisation dans notre méta-modèle. Grâce aux relations d'organisation, on peut créer des diagrammes de patrons qui représentent les liens de composition, de raffinement et de variation entre des patrons. Les diagrammes de patrons peuvent faciliter la sélection d'un patron convenable à réutiliser[3].

La Figure II-46 montre les relations d'organisation de patrons définies dans notre méta-modèle.

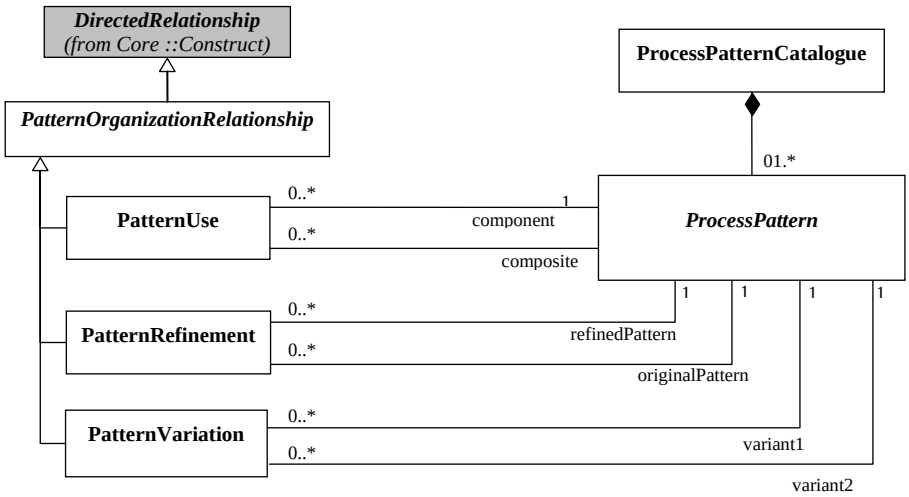


Figure II-46. Relations d'organisation de patrons

PatternUse

La relation *PatternUse* existe entre deux patrons si le problème d'un patron (*component*) est un sous-problème de celui de l'autre patron *(composite)*.

---

[1] FINISH est l'action de terminaison d'un processus.

[2] Différents termes peuvent être employés pour désigner un tel système, par exemple «catalogue », « famille » ou « langage » de patrons. Il s'agit dans tous les cas d'une collection de patrons couvrant un même domaine, et dont l'application est assurée par des règles qui permettent de les combiner

[3] Surtout dans le cas où il n'existe pas de patron qui satisfasse exactement le besoin de réutilisation, les diagrammes de patrons peuvent guider le concepteur à sélectionner le patron le plus approprié possible.

PatternRefinement

La relation *PatternRefinement* existe entre deux patrons si le problème d'un patron (*refinedPattern*) est un raffinement de celui d'un autre (*orginalPattern*).

PatternVariation

La relation *PatternVariance* associe deux patrons (*variant1* et *variant2*) s'ils résolvent un même problème en utilisant différentes solutions.

Règles de bonne modélisation

La sémantique de ces relations est précisée par les contraintes suivantes :

[C51] Une relation *PatternUse* entre deux patrons signifie que le problème du modèle composant doit être un sous-problème du modèle composé.

```
context PatternUse inv:
self.composite.problem.subproblem
    → includes(self.component.problem)
```

[C52] Si deux patrons sont reliés par une relation *PatternRefinement*, cela signifie que le problème d'un patron est une spécialisation du problème de l'autre.

```
context PatternRefinement inv:
self.originalPattern.problem.refinedproblem →
              includes(self.refinedPattern.problem)
```

[C53] Si deux patrons sont reliés par une relation *PatternAlternative*, cela signifie que ces patrons adressent le même problème mais proposent des solutions différentes.

```
context PatternAlternative inv:
let A = self.variant1 B = self.variant2
A.problem = B.problem and
A.context.resultingcontext = B.context.resultingcontext
and A.solution <>B.solution
```

Les patrons de procédé peuvent être groupés et stockés dans des catalogues de patrons (*ProcessPatternCatalogue*).

Notation

Le Tableau II-10 montre les notations représentant les relations d'organisation des patrons de procédé.

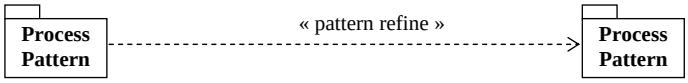
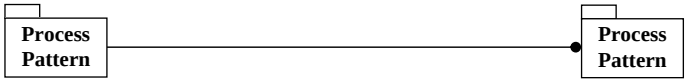
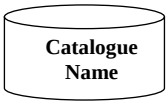
UML-PP Méta-classe	UML Classe de base	Notation
PatternUse	Directed Relationship	
Pattern Refinement	Directed Relationship	
Pattern Variation	Directed Relationship	
ProcessPattern Catalogue	Classifier	

Tableau II-10. Notations des relations d'organisation de patrons de procédé

Nous montrons dans la Figure II-47 des exemples de relations d'organisation de patrons de procédé.

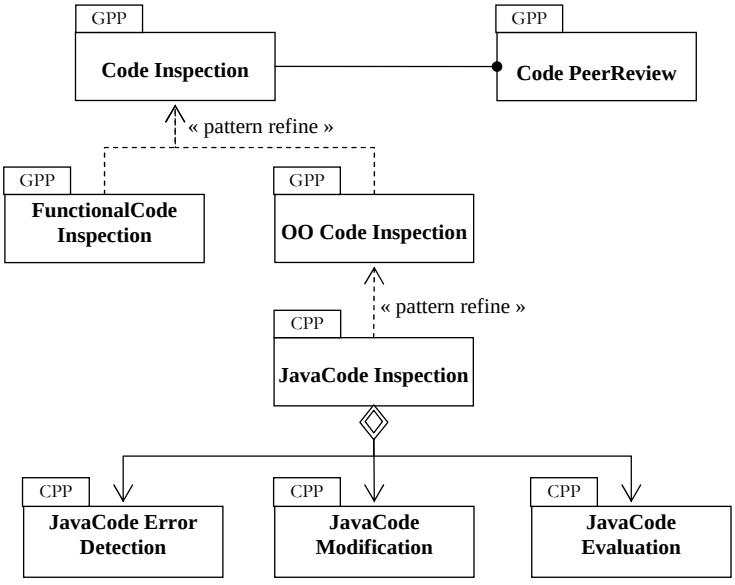


Figure II-47. Exemples de relations d'organisation entre patrons de procédé

Nous voyons dans cet exemple un ensemble de patrons de procédé concernant le problème de la vérification de code. *Code Inspection* est un patron général dont les raffinements *FunctionalCode Inspection* et *OO CodeInspection* sont aussi généraux. À son tour, *OOCode Inspection* est raffiné sous forme du patron concret *JavaCode Inspection*. Le patron *JavaCode Inspection* se compose de trois autres patrons : *JavaCode Error Detection*, *JavaCode Modification* et *JavaCode Evaluation*. L'exemple montre aussi la relation de variation entre *Code Inspection* et *Code Review* qui sont deux patrons représentant deux méthodes différentes pour vérifier un code.

III. CONCLUSION

Dans ce chapitre nous avons présenté le méta-modèle UML-PP permettant la définition et la représentation de procédés à base de patrons réutilisables.

Nous avons défini les concepts nécessaires pour représenter les éléments primaires de procédé et les patrons de procédé. En particulier, nous permettons de représenter les modèles de procédé et les patrons de procédé à différents niveaux d'abstraction pour traduire la diversité des connaissances de procédé.

Nous avons également défini deux types de relations représentant la manière d'appliquer des patrons de procédé pour définir un élément de procédé ou pour réorganiser un ensemble d'éléments de procédé. En permettant la paramétrisation de ces relations, nous promouvons la réutilisation de patrons à différents niveaux d'abstraction. La sémantique de ces relations sera formalisée dans le prochain chapitre, dans l'objectif de permettre leur automatisation. Cependant, la relation *Applying* est sujette à discussion dans la mesure où sa mise en œuvre nécessite la résolution de conflits structurels entre le patron source et le modèle cible. Par ailleurs, pour refléter les besoins de modélisations les plus courants, nous avons introduit trois modes d'application de cette relation ; il faudrait peut-être réfléchir à d'autres modes pour couvrir toutes les situations. Ces travaux entrent dans les perspectives de cette thèse.

CHAPITRE III.

MÉTHODE DE MODÉLISATION DE PROCÉDÉS À BASE DE PATRONS RÉUTILISABLES

Après avoir formalisé le concept de patron de procédé (chapitre II), nous voulons maintenant fournir des moyens pour, d'une part, appliquer de manière systématique des patrons de procédé et, d'autre part, guider leur réutilisation dans la modélisation de procédés logiciels.

Dans ce chapitre nous proposons une méthode de réutilisation de patrons de procédé pour construire ou pour améliorer des modèles de procédé. Nous présentons tout d'abord les objectifs de notre approche (section I), puis nous définissons un ensemble d'opérateurs de base permettant de manipuler les patrons de procédé (section II). Enfin, nous décrivons un méta-procédé pour modéliser les procédés logiciels par réutilisation de patrons (section III). Ce méta-procédé est bien sûr conforme au méta-modèle UML-PP décrit dans le chapitre précédent.

I. PRINCIPE DE NOTRE APPROCHE

Dans cette section, nous introduisons notre proposition sur l'utilisation de patrons de procédé pour pouvoir réduire l'effort de modélisation de procédés et augmenter la qualité des procédés modélisés. Nous mettons d'abord en évidence le besoin d'une méthode, puis nous précisons la terminologie employée pour faciliter la lecture de la suite de ce chapitre.

I.1. BESOIN D'UNE MÉTHODE DE MODÉLISATION DE PROCÉDÉS À BASE DE PATRONS

S'il existe de nombreuses méthodes de développement des produits logiciels, il n'en est pas de même pour le développement des procédés logiciels. Il existe plusieurs méthodes d'évaluation et d'amélioration de procédés, mais peu de méthodes dédiées à la modélisation de procédés (c.f. section I.4.2 du Chapitre I). Les méthodes disponibles ne répondent pas complètement aux besoins des concepteurs de procédés : les démarches qu'elles proposent sont souvent informelles et présentées avec un niveau de détail insuffisant. Elles se limitent pour la plupart à suggérer une organisation du cycle de vie en étapes globales, et ne permettent pas un guidage fin des activités de modélisation de procédés. De plus, elles ne tiennent pas suffisamment compte de la réutilisation de connaissances de procédé.

Les méthodes de modélisation de procédés à base de patrons réutilisables, quant à elles, n'en sont qu'à leur début (c.f. section III.4 du Chapitre I). Les patrons de procédé sont encore peu réutilisés à cause du manque de moyens efficaces pour les sélectionner en fonction du problème à résoudre, pour les appliquer et les adapter en fonction du contexte.

Ces limites nous convainquent qu'il est nécessaire de disposer d'une méthode de modélisation de procédés à base de patrons réutilisables. Pour fournir une aide efficace aux concepteurs de procédés, une telle méthode devra être fondée sur un procédé rigoureux et (semi)automatisable permettant de générer des modèles de procédé en réutilisant autant que possible des patrons de procédé.

Nous proposons donc une méthode de modélisation de procédés à base de patrons réutilisables comportant deux éléments complémentaires :

- un ensemble d'opérateurs de réutilisation permettant de manipuler les patrons de procédé dans le but de les réutiliser.
- un méta-procédé décrivant quand et comment réutiliser les patrons pour élaborer des modèles de procédé.

Plus précisément, le méta-procédé définit un ensemble de méta-tâches pour modéliser les procédés. En s'exécutant, ces méta-tâches peuvent utiliser les opérateurs de réutilisation. La Figure III-1 montre la relation entre les éléments constituants de notre méthode.

Figure III-1. Éléments constituants de notre méthode de modélisation par réutilisation

Dans le but de fournir une assistance outillée aux concepteurs, nous souhaitons développer un environnement de modélisation guidé par le méta-procédé, et fondé sur des opérateurs de réutilisation automatisés. Pour cela, nous devons définir en détail le méta-procédé ainsi que la sémantique précise des opérateurs.

Nous présentons la définition des opérateurs dans la section II, et le méta-procédé dans la section III.

I.2. TERMINOLOGIE

Comme nous l'avons remarqué dans l'état de l'art, il y a encore peu de travaux sur les méthodes de modélisation de procédés incluant l'utilisation de patrons de procédé. La terminologie du domaine n'est donc pas encore stabilisée. Pour faciliter la compréhension de ce chapitre, nous précisons ici la signification des termes utilisés.

Méta-procédé

> Un méta-procédé est un procédé définissant les *activités* à réaliser pour élaborer ou modifier les modèles de procédé.

Méta-tâche

> Une méta-tâche est une activité contrôlée du méta-procédé.

Imitation de patron de patron de procédé [1]

> L'imitation d'un patron est la duplication du modèle de procédé capturé dans la solution du patron pour l'intégrer (en l'adaptant éventuellement) au modèle de procédé à élaborer ou modifier.

Opérateurs de réutilisation de patrons de procédé

> Les opérateurs de réutilisation de patrons de procédé représentent des opérations pour sélectionner un patron répondant à un problème donné, et pour imiter sa solution dans un contexte spécifique.

[1] Pour nous, l'application d'un patron de procédé est l'action de réutilisation de la solution du patron pour résoudre un problème donné. Le processus d'application peut être tacite surtout quand la solution du procédé n'est pas formalisée. L'imitation d'un patron de procédé est un cas particulier de l'application d'un patron où le processus d'application manipule explicitement la solution du patron représentée sous forme de modèle. Certains travaux utilisent le terme «instancier» en parlant de la réutilisation du modèle capturé dans la solution d'un patron. Nous pensons que ce terme ne reflète pas exactement la nature de l'action car le modèle réutilisé reste au même niveau d'abstraction que le modèle originel du patron. Nous adoptons donc le terme d'"imitation' proposé dans [Rieu99].

Opérateurs d'adaptation de patrons de procédé

> Les opérateurs d'adaptation de patrons de procédé représentent des opérations de modification de la solution d'un patron (qui est un modèle de procédé) [1] pour un besoin spécifique. L'adaptation peut être le renommage, la spécialisation, l'addition ou la suppression d'éléments du modèle de procédé.

II. OPÉRATEURS DE RÉUTILISATION

Dans cette section, nous décrivons la formalisation des opérateurs de réutilisation et d'imitation. Nous définissons tout d'abord ces opérateurs (section II.1) puis nous en donnons une sémantique opérationnelle (section II.2) pour les rendre exécutables.

II.1. DÉFINITION DES OPÉRATEURS DE RÉUTILISATION

Pour faciliter la réutilisation de patrons de procédé, il faut permettre une automatisation maximale de leur manipulation. Dans ce but, nous proposons des opérateurs de base pour manipuler les patrons de procédé. Ces opérateurs sont classés en trois catégories : *opérateurs de recherche*, *opérateurs d'adaptation* et *opérateurs d'imitation* (Figure III-2).

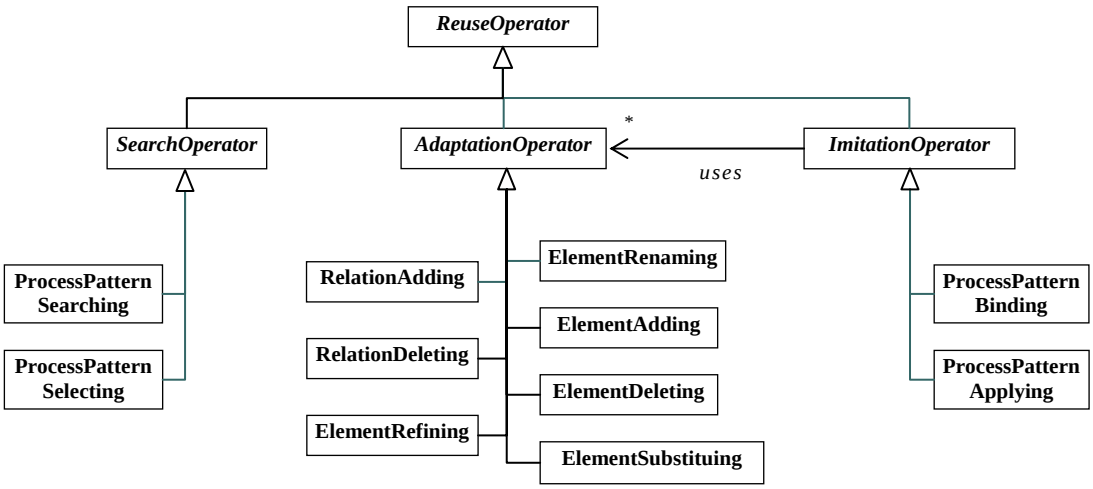


Figure III-2. Opérateurs de réutilisation

Dans cette section, nous utilisons une approche dénotationnelle pour spécifier les opérateurs de réutilisation de patrons de procédé. Plus précisément, nous décrivons pour chaque opérateur ses opérandes et le résultat obtenu après son exécution.

Le Tableau III-1 présente les notations communes utilisées dans les descriptions de ces opérateurs.

[1] En fait, ces opérateurs sont des opérateurs de manipulation de modèles de procédé.

Notation	Signification
N	Un ensemble d'identifiants valables pour les éléments de modélisation
TE	Un ensemble de types d'éléments de procédé. Un élément $te \in TE$ peut avoir une valeur parmi « produit », « tâche » ou « rôle».
TR	Un ensemble de types de relations entre des éléments de procédé. Un élément $tr \in TE$ peut avoir, par exemple, la valeur « aggregation», «refinement», ou « taskprecedence».
E	Un ensemble d'éléments de procédé. Un élément $e \in E$ a un identifiant noté $e_{name} \in N$, un type noté $e_{type} \in TE$
R	Un ensemble de relations définies entre des éléments de procédé. Un élément $r \in R$ peut être une composition, généralisation ou relation spéciale définie entre deux types d'éléments de procédé spécifiques (par exemple relation entre un rôle et une tâche, entre deux tâches, etc.). Une relation est définie par un identifiant, deux éléments qu'elle relie[1] aussi qu'un type de relation. On dénote : - $r_{name} \in N$: l'identifiant de la relation r - $r_{source} \in E$: l'élément source de la relation r - $r_{dest} \in E$: l'élément destination de la relation r - $r_{type} \in TR$: le type de la relation r
PM	Un ensemble de modèles de procédé. Un élément $pm \in PM$ est défini par (E_{pm} , R_{pm}): - E_{pm} : l'ensemble d'éléments contenus dans pm - R_{pm} : l'ensemble de relations définies entre des éléments de E_{pm}
P	Un ensemble de problèmes de modélisation de procédés
C	Un ensemble de contextes d'utilisation de patrons de procédé. Un élément $c \in C$ est défini par un état initial, un état résultant et une situation d'application. On dénote : - $c_{initial}$: l'état initial du contexte c - $c_{resulting}$: l'état résultant du contexte c - $c_{application}$: la situation d'application recommandée par le contexte c
PP	Un ensemble de patrons de procédé. Un élément $pp \in PP$ est défini par un problème, une solution et un contexte. On dénote : - $pp_{problem}$: le problème du patron, $pp_{problem} \in P$ - $pp_{solution}$: la solution du patron, $pp_{solution} \in PM$ - $pp_{context}$: le contexte du patron, $pp_{context} \in C$
$\succ$[2]	Une relation de raffinement. $e_1 \succ e_2$ signifie que e_2 est un raffinement de e_1

Tableau III-1. Notations utilisées pour formaliser les opérateurs de réutilisation

II.1.1. Opérateurs de recherche

Les opérateurs de recherche de patrons permettent, à partir d'un ensemble de patrons de procédé, d'identifier et de sélectionner les patrons les plus adaptés à un problème de

[1] Nous définissons les relations entre les éléments de procédé en tant que relations directes (c.f. Chapitre 2).

[2] Pour simplifier la représentation, nous n'utilisons ici qu'une seule notation pour toutes les relations de raffinement. Pour la signification des différents types de raffinement (entre problèmes, éléments de procédé, ou patrons), c.f le Chapitre II.

modélisation de procédés donné. Nous proposons deux opérateurs dans cette catégorie : *ProcessPatternSearching* et *ProcessPatternSelecting*.

ProcessPatternSearching

L'opérateur *ProcessPatternSearching* a pour but de chercher, dans un ensemble de patrons de procédé (une base de patrons), des patrons dont l'intention correspond [1] au problème exprimé.

$$ProcessPatternSearching : P \times PP \rightarrow PP$$

$$ProcessPatternSearching(p,BP) = \{pp \in BP \mid (pp_{problem} = p) \vee (pp_{problem} \succ p)\}$$

Dans cette formule :

- $p \in P$ est le problème à résoudre, $BP \subseteq PP$ est la base de patrons de procédé.

ProcessPatternSelecting

L'opérateur *ProcessPatternSelecting* a pour but de sélectionner dans une base de patrons de procédé les patrons les plus adaptés au contexte indiqué.

$$ProcessPatternSelecting : C \times PP \rightarrow PP$$

$$ProcessPatternSelecting(c, BP) = \{pp \in BP \mid Conformity(pp_{context}, c) = max\}$$

Dans cette formule :

- $c \in C$ est le contexte demandé, $BP \in PP$ est la base de patrons de procédé
- *max* est la valeur du degré de conformité maximum obtenu.

Cet opérateur nécessite la fonction *Conformity(c1,c2)* pour mesurer la conformité entre le contexte d'un patron examiné *(c1)* et le contexte demandé *(c2)*. D'une manière basique, les degrés de conformité sont définis comme suit :

/* c1 et c2 identiques */

$$Conformity = 6,\ (c1_{initial}=c2_{initial}) \wedge (c1_{resulting}=c2_{resulting}) \wedge (c1_{application}=c2_{application})$$

/* résultat de c1 et c2 identiques, conditions initiales et applicatives différentes */

$$Conformity = 5,\ (c1_{initial}=c2_{initial}) \wedge (c1_{resulting}=c2_{resulting}) \wedge (c1_{application} \succ c2_{application})$$

$$Conformity = 4,\ (c1_{initial} \succ c2_{initial}) \wedge (c1_{resulting}=c2_{resulting}) \wedge (c1_{application} \succ c2_{application})$$

/* c2 plus spécifique que celui de c1, conditions initiales et applicatives différentes */

$$Conformity = 3,\ (c1_{initial} \succ c2_{initial}) \wedge (c1_{resulting} \succ c2_{resulting}) \wedge (c1_{application} \succ c2_{application})$$

$$Conformity = 2,\ (c1_{initial} \succ c2_{initial}) \wedge (c1_{resulting} \succ c2_{resulting}) \wedge (c1_{application} \neq c2_{application})$$

$$Conformity = 1,\ (c1_{initial} \neq c2_{initial}) \wedge (c1_{resulting} \succ c2_{resulting}) \wedge (c1_{application} \neq c2_{application})$$

/* c1 et c2 différents */

$$Conformity = 0,\ (c1 \neq c2)$$

En réalité la comparaison entre contextes est beaucoup plus complexe car il faut examiner la similarité entre plusieurs éléments de contexte qui peuvent être décrits informellement.

[1] C'est-à-dire que soit le problème adressé par un patron choisi est identique au problème à résoudre, soit le problème à résoudre est un raffinement du problème du patron (c.f. Chapitre 2).

L'automatisation totale et efficace d'une telle opération n'est pas réaliste et déborde du cadre de cette thèse.

Pour l'instant, nous supposons que la fonction *Conformity* est définie par les règles simples ci-dessus. Une discussion sur la sémantique opérationnelle de cette fonction est présentée plus loin dans la section II.2.

Pour illustrer la signification des opérateurs de recherche, nous donnons un exemple ci-dessous (Figure III-3). Supposons l'existence d'une base de patrons de procédé, nommée *QualityControlPBase*, qui contiennne différents patrons adressant des problèmes concernant les activités de contrôle de qualité d'artéfacts logiciels.

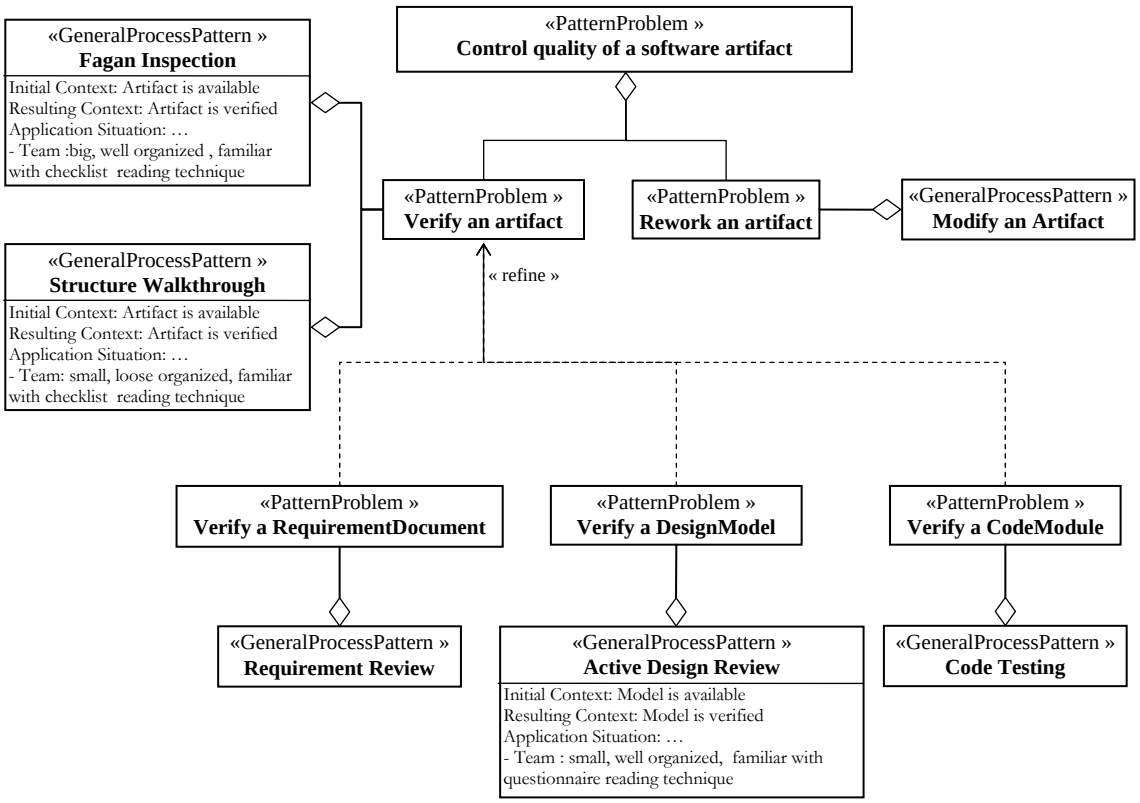


Figure III-3. Base de patrons de procédé *QualityControlBase*

Lorsqu'un concepteur de procédé cherche un patron convenable pour vérifier un modèle de conception, il applique d'abord l'opérateur *ProcessPatternSearching* :

```
(1) R1 = ProcessPatternSearching («Verify a Design Model», QualityControlBase)
```

Après l'exécution de (1), R1 contient trois patrons correspondant au problème « Verify a Design Model » : `R1 = {ActiveDesignReview, FaganInspection, StructuredWalkthrough}`

Pour sélectionner le patron le plus adapté à son équipe, il applique ensuite l'opérateur *ProcessPatternSelecting* :

```
(2) R2 = ProcessPatternSelecting («small team, familiar with questionnaire
    reading technique», R1)
```

Parmi ces trois patrons, seul *ActiveDesignReview* répond exactement au problème (« Verify a DesignModel»), les deux autres adressant le problème plus général : « Verify an artifact ». De plus, les contextes des deux patrons *FaganInspection* et *Structure Walkthrough* préconisent la technique « Cheklist » pour relire tous les types d'artéfact, alors que pour examiner les modèles de conception, le patron *ActiveDesignReview* préfère la technique à base de questionnaires qui est supposée être maîtrisée par l'équipe du conception. Après l'exécution de (2), R2 contient donc :

```
R2 = {ActiveDesignReview}
```

II.1.2. Opérateurs d'adaptation

Un patron est rarement appliqué sans modification. En général, l'application nécessite une adaptation de la solution du patron à un contexte spécifique. Cela peut être réalisé grâce aux opérateurs d'adaptation qui permettent de manipuler les modèles de procédé capturés dans les solutions de patrons. Ces opérateurs servent aux opérateurs d'imitation de patrons. Dans cette catégorie, nous proposons les opérateurs suivants : *ElementRenaming*, *ElementAdding*, *ElementDeleting*, *RelationAdding, RelationDeleting, ElementSubstituing* et *ElementRefining*.

En fait, ces opérateurs ne manipulent pas les patrons de procédé, mais les modèles capturés dans leur solution. Autrement dit, ce sont des opérateurs de manipulation de modèles spécialement adaptés au contexte des modèles de procédé.

ElementRenaming

L'opérateur *ElementRenaming* réalise une substitution lexicale du nom d'un élément à un autre[1].

$$ElementRenaming : E \times N \rightarrow E$$

$$ElementRenaming(e,n) = e \mid (e_{name} = n)$$

Dans cette formule :

- $e \in E$ est l'élément de procédé à renommer
- $n \in N$ est le nouveau identifiant pour e.

ElementAdding

L'opérateur *ElementAdding* permet d'ajouter un nouvel élément à un modèle de procédé.

$$ElementAdding : PM \times E \rightarrow PM$$

$$ElementAdding(pm,e) = pm \mid E_{pm} = (E_{pm} \cup e)$$

Dans cette formule :

- $pm \in PM$ est le modèle de procédé à enrichir.
- $e \in E$ est le nouvel élément de procédé à ajouter

[1] Le renommage d'un élément dans un modèle peut avoir un impact sur des éléments reliés. Cet effet est traité dans le contexte de l'action qui utilise l'opérateur *ElementRenaming*.

ElementDeleting

L'opérateur *ElementDeleting* permet de supprimer un élément d'un modèle de procédé (et les relations auxquelles il participe)[1].

$$ElementDeleting : PM \times E \rightarrow PM$$

$$ElementDeleting(pm,e) = pm \mid (E_{pm} \setminus e) \wedge ((R_{pm} \setminus \{r \mid (r_{source}=e) \wedge ((r_{dest}=e)\})$$

Dans cette formule :

- $pm \in PM$ est le modèle de procédé à simplifier
- $e \in E_{pm}$ est l'élément de procédé à enlever

RelationAdding

L'opérateur *RelationAdding* permet d'ajouter une nouvelle relation entre deux éléments d'un modèle de procédé.

$$RelationAdding : E \times E \times TR \times PM \rightarrow PM$$

$$RelationAdding(e1,e2,tr,pm) = pm \mid R_{pm} = ((R_{pm} \cup r) \wedge (r_{source}=e_1) \wedge (r_{dest}=e_2 \wedge (r_{type} = tr))$$

Dans cette formule :

- $pm \in PM$ est le modèle de procédé à enrichir.
- $e1,e2 \in E_{pm}$ sont les éléments à connecter
- $tr \in TR$ est le type de la relation à établir entre *e1* et *e2*

RelationDeleting

L'opérateur *RelationDeleting* permet de supprimer une relation entre deux éléments d'un modèle de procédé.

$$ElementDeleting : E \times E \times TR \times PM \rightarrow PM$$

$$ElementDeleting(e1,e2,tr,pm) = pm \mid R_{pm} = (R_{pm} \setminus \{r \mid (r_{source}=e1) \wedge (r_{dest}=e2) \wedge (r_{type} = tr)\})$$

Dans cette formule :

- $pm \in PM$ est le modèle de procédé à simplifier
- $e1, e2 \in E_{pm}$ sont les éléments à déconnecter
- $tr \in TR$ est le type de la relation à supprimer entre *e1* et *e2*.

ElementSubstituting

L'opérateur *ElementSubstituting* permet de remplacer un élément d'un modèle par un autre élément qui n'est pas déjà contenu dans le modèle. Les relations définies sur l'élément originel doivent être réorientées vers le nouvel élément.

$$ElementSubstituing : E \times E \times PM \rightarrow PM$$

$$ElementSubstituing(e1,e2,pm) = pm \mid (E_{pm} = (E_{pm} \setminus e1) \cup e2)) \wedge ((R_{pm} = (R_{pm} \setminus \{r1\}) \cup \{r2\})$$
$$\mid ((r1_{source}=e1) \vee ((r1_{dest}=e1)) \wedge (((r2_{source}=e2) \vee ((r2_{dest}=e2))$$

[1] La suppression d'un élément peut causer la suppression des éléments qui lui sont reliés par les relations d'agrégation. Cet effet est traité dans le contexte de l'action qui utilise l'opérateur *ElementDeleting*

Dans cette formule :

- $e_1 \in E_{pm}$ est l'élément à substituer ; $e1 \notin E_{pm}$ est le substitut
- $pm \in PM$ est le modèle comportant l'élément à substituer

ElementRefining

L'opérateur *ElementRefining* permet de raffiner un élément d'un modèle par un autre élément plus spécifique. Autrement dit, c'est une substitution dans laquelle le remplaçant doit être un raffinement de l'élément remplacé.

$$ElementRefining : E \times E \times PM \rightarrow PM$$

$$ElementRefining(e1,e2,pm) = ElementSubstituing(e1,e2,pm) \wedge (e1 \succ e2)$$

Dans cette formule :

- $e1 \in E_{pm}$ est l'élément à raffiner
- $e2 \in E_{pm}$ est l'élément plus spécifique jouant le rôle de remplaçant
- $pm \in PM$ est le modèle comportant l'élément à raffiner

II.1.3. Opérateurs d'imitation

Les opérateurs d'imitation de patrons permettent de produire le modèle résultant de l'application d'un patron à un élément ou à un ensemble d'éléments de procédé. Nous proposons deux opérateurs correspondant aux deux relations d'application de patrons de procédé définies dans le Chapitre 2 : *ProcessPatternBinding* et *ProcessPatternApplying* [1].

Comme nous l'avons expliqué dans le méta-modèle, nous utilisons le mécanisme d'abstraction pour réutiliser les patrons de procédé. Pour cela, nous utilisons les patrons paramétrés et leur spécialisation en appliquant des règles de substitution.

Nous ajoutons quelques nouvelles notations (Tableau III-2) pour représenter ce mécanisme dans la définition des opérateurs d'imitation.

Notation	Signification
SR	Un ensemble de règles de substitution d'éléments de procédé. Un élément $s(pformal/pactual) \in SR$ comporte deux constituants : - $s_{pformal} \in E$: élément jouant le rôle de paramètre formel de la substitution - $s_{pactual} \in E$: élément jouant le rôle de paramètre effectif de la substitution - On dénote : $SR_{pformal}$ ensemble de paramètres formels de SR, $SR_{pactual}$ ensemble de paramètres effectifs de SR
~	Une relation de compatibilité. $e_1 \sim e_2$ signifie que le type de e_1 est compatible avec celui de e_2
←	Une relation de description définie sur E x PM. $e \leftarrow pm$ signifie que l'élément e est décrit par le modèle pm

Tableau III-2. Notations supplémentaires pour décrire les opérateurs d'imitation

[1] Dans le méta-modèle UML-PP, les concepts de *ProcessPatternBinding* et *ProcessPatternApplying* sont définis en tant qu'éléments de modélisation (relations). Dans ce chapitre dont l'objectif est la mise en œuvre de procédés, nous les présentons comme des opérateurs exécutables.

ProcessPatternBinding

L'opérateur *ProcessPatternBinding* a pour but de générer un modèle décrivant un élément de procédé à partir de la solution d'un patron de procédé.

Le résultat de cet opérateur est obtenu par duplication du modèle de la solution du patron, et éventuellement en substituant certains éléments de ce modèle avec d'autres plus spécifiques décrivant l'élément à définir.

$$ProcessPatternBinding : E \times PP \times SR \rightarrow PM$$

$$ProcessPatternBinding(pe,pp,sub) = pm \mid (e \leftarrow pm) \wedge (E_{pm} = (E_{pp_{solution}} \setminus \{e \in sub_{pformal}\}) \cup sub_{pactual}) \wedge (R_{pm} = R_{pp_{solution}} \setminus \{r1\}) \cup \{r2\}) \mid ((r1_{source} \vee r1_{dest}) \in sub_{pformal})) \wedge ((r2_{source} \vee r2_{dest}) \in sub_{pactual})$$

Dans cette formule :

- $pe \in E$ est l'élément à définir
- $pm \in PM$ est le modèle à élaborer pour décrire pe
- $pp \in PP$ est le patron correspondant à la spécification de l'élément pe
- $sub \in SR$ est l'ensemble des règles de substitution de paramètres. Il doit satisfaire les conditions suivantes :
  (1) Les paramètres formels décrits dans les règles de substitution doivent être des éléments du modèle de la solution du patron.
  $$sub_{pformal} \subseteq E_{pp_{solution}}$$
  (2) Les paramètres effectifs doivent être compatibles avec les paramètres formels
  $$sub_{pactual} \sim sub_{pformal}$$

ProcessPatternApplying

L'opérateur *ProcessPatternApplying* est utilisé pour appliquer la solution d'un patron de procédé à un modèle de procédé pour l'enrichir ou le (re)structurer.

Le modèle résultant de cet opérateur est obtenu grâce à une fusion de deux modèles : le modèle de la solution du patron et le modèle de procédé à enrichir ou restructurer. En particulier, la substitution de paramètres est possible en fusionnant ces modèles.

$$ProcessPatternApplying : PP \times PM \times SR \rightarrow PM$$

$$ProcessPatternApplying(pp,pm,sub) = pm \mid (E_{pm} = (E_{pp_{solution}} \cup E_{pm}) \setminus \{e \in sub_{pformal}\})) \wedge (R_{pm} = R_{pp_{solution}} \cup R_{pm} \setminus \{r\} \mid ((r_{source} \vee r_{dest}) \in sub_{pformal})$$

Dans cette formule :

- $pp \in PP$ est le patron à appliquer
- $pm \in PM$ est le modèle à enrichir ou (re)structurer
- $sub \in SR$ est l'ensemble des règles de substitution de paramètres. Il doit satisfaire les conditions suivantes :

(1) Les paramètres formels décrits dans les règles de substitution doivent être des éléments du modèle de la solution du patron.

$$sub_{pformal} \subseteq E_{pp_{solution}}$$

(2) Les paramètres effectifs décrits dans les règles de substitution doivent être des éléments du modèle originel.

$$sub_{pactual} \subseteq E_{pm}$$

(3) Les paramètres effectifs doivent être compatibles avec les paramètres formels

$$sub_{pactual} \sim sub_{pformal}$$

Pour illustrer la mise en oeuvre des opérateurs d'imitation, nous donnons un exemple simple. La Figure III-4 montre le modèle de procédé *Information System Development* qui comporte trois tâches générales représentant le développement de trois modules d'un système d'information.

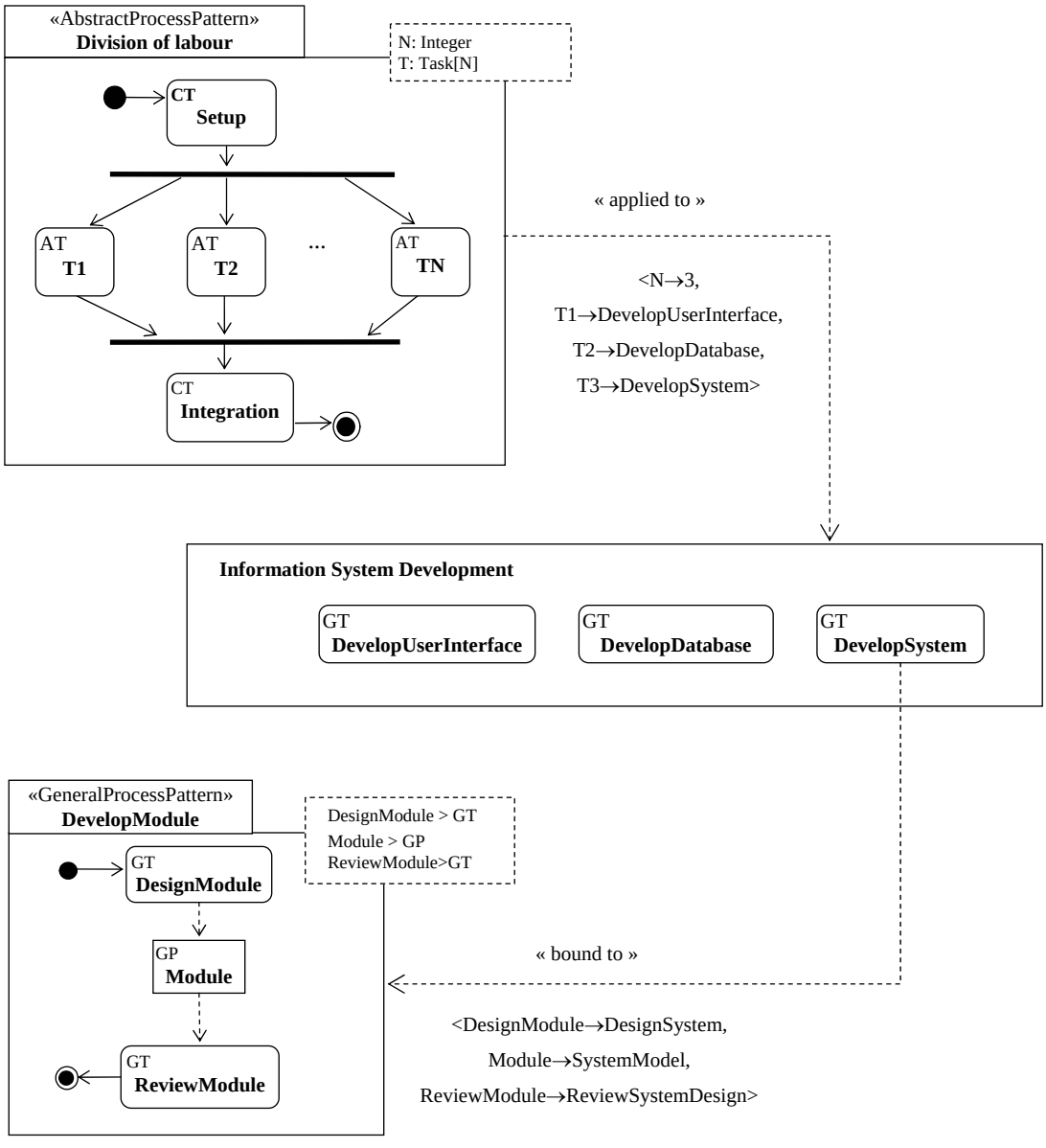


Figure III-4. Modèle de procédé basé sur la réutilisation de patrons de procédé

Pour raffiner ce modèle, le concepteur réutilise deux patrons : le patron abstrait *Division of labour,* décrivant une structure de procédé composée d'activités parallèles, est appliqué pour organiser les éléments du modèle *Information System Development* (via la relation *ProcessPatternApplying*) ; le patron général *DevelopModule,* décrivant les activités à faire pour construire un module d'un système, est appliqué pour définir le contenu de l'élément *DevelopSystem* (via la relation *ProcessPatternBinding*).

Pour générer le modèle résultant du modèle de la Figure III-4, l'opérateur *ProcessPatternApplying* suivant est tout d'abord exécuté :

```
(1) ProcessPatternApplying(Division of Labour, Information System Development,
                {T1/DevelopUserInterface,T2/DevelopDatabase,T3/DevelopSystem})
```

Le résultat de l'opérateur (1) est montré dans la Figure III-5 a. Cet opérateur ajoute les éléments du modèle du patron *Division of labour* au modèle *Information System Development* en remplaçant les tâches abstraites jouant les rôles de paramètres formels par les tâches du modèle *Information System Development*.

Ensuite, pour générer le contenu concret de la tâche *DevelopSystem*, l'opérateur *ProcessPatternBinding* suivant est exécuté :

```
(2) ProcessPatternBinding(DevelopSystem, DevelopModule,
                        {DesignModule/DesignSystem, Module/SystemModel,
                                ReviewModule/ReviewSystemDesign})
```

Le résultat de l'opérateur (2) est montré dans la Figure III-5 b. La tâche *DevelopSystem* est maintenant décrite par un modèle comportant des tâches et des produits concrets.

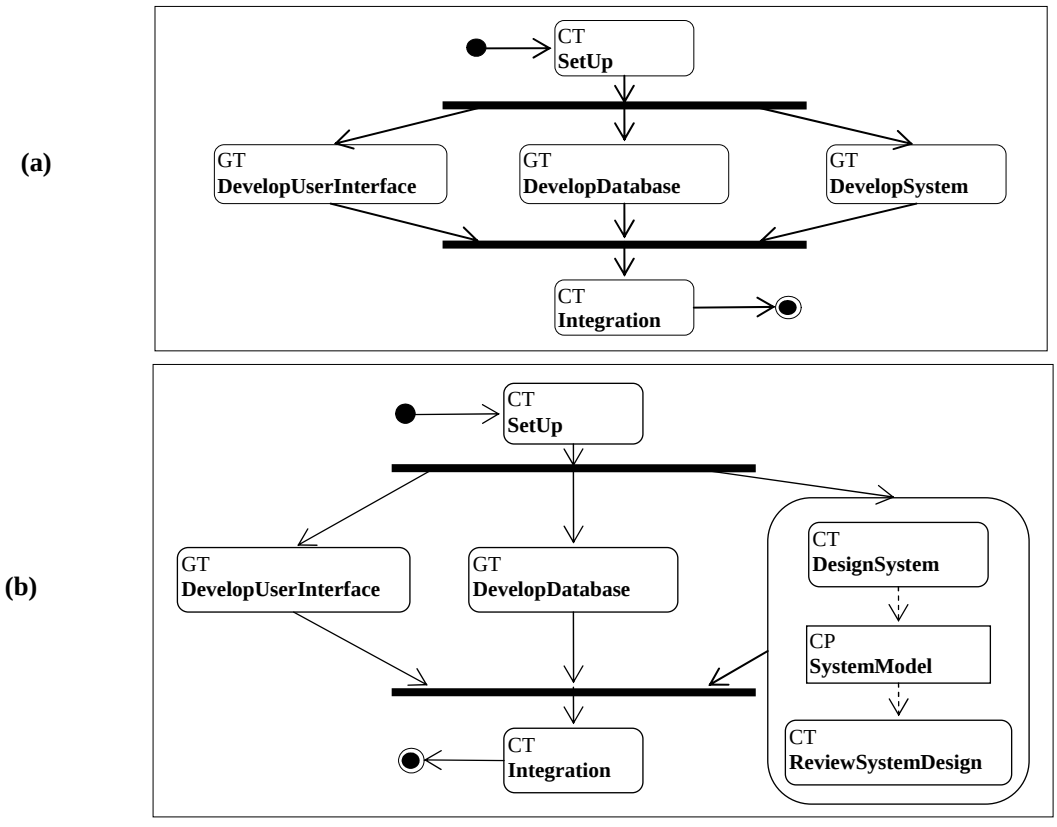


Figure III-5. Résultat d'exécution des opérateurs d'imitation de la Figure III-4

La définition d'opérateurs représentée ci-dessus ne peut spécifier que les contraintes concernant l'état du modèle avant et après l'exécution d'un opérateur. L'approche dénotationnelle peut être utilisée pour spécifier (de manière déclarative) le comportement d'opérateurs, sans toutefois pouvoir exprimer impérativement des modifications de modèles. Par conséquent, elle ne suffit pas pour l'implémentation des opérateurs proposés. Visant une automatisation des opérateurs de réutilisation de patrons de procédé, nous définissons dans la suite leur sémantique opérationnelle.

II.2. SÉMANTIQUE OPERATIONNELLE DES OPÉRATEURS DE RÉUTILISATION

Cette section est consacrée à la sémantique opérationnelle des opérateurs de réutilisation. Nous définissons ces opérateurs sous forme de méta-opérations associées à des concepts du méta-modèle UML-PP (Figure III-6).

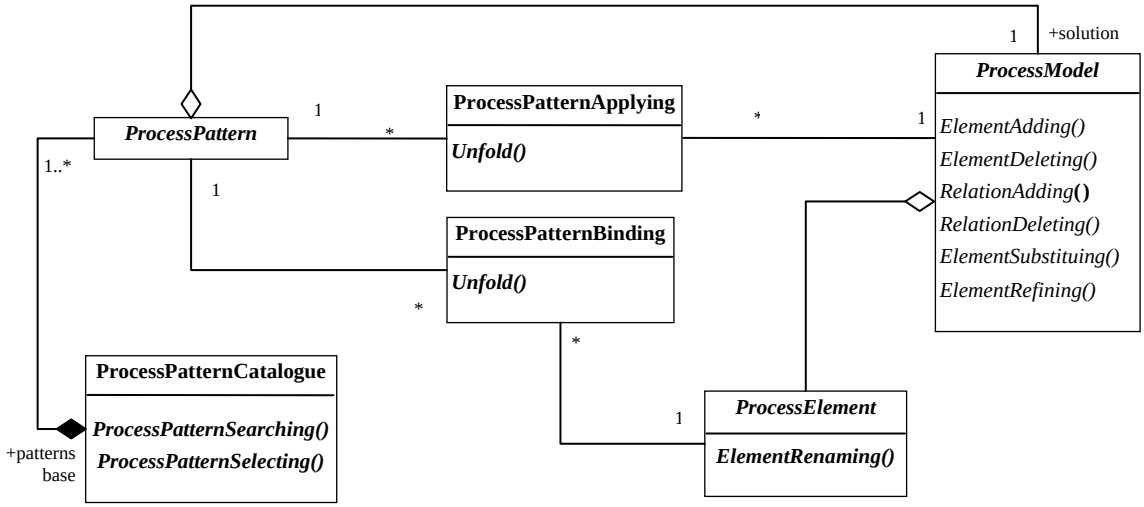


Figure III-6. Implémentation d'opérateurs sous forme de méta-opérations

Les opérateurs de recherche de patrons souhaités dans un ensemble de patrons sont associés à la méta-classe *ProcessPatternCatalogue*. Les opérateurs d'adaptation s'exécutent sur les éléments et les modèles de procédé, ils sont donc définis comme des méta-opérations des méta-classes *ProcessModel* et *ProcessElement*. Quant aux opérateurs d'imitation, ils sont définis dans notre méta-modèle en tant que relations *ProcessPatternBinding* et *ProcessPatternApplying*. Nous ajoutons à chacune de ces relations la méta-opération *Unfold()* pour réaliser la fonction de l'opérateur.

Dans le cadre de cette thèse, nous ne définissons que la sémantique opérationnelle des opérateurs de recherche et d'imitation. Les opérateurs d'adaptation étant des opérateurs de manipulation de modèles, la définition de leur sémantique opérationnelle n'est pas un objectif essentiel de notre travail. Par ailleurs, il existe déjà plusieurs travaux sur ce sujet, par exemple [Ribo02][Bernstein03][Kurtev06][Reddy06].

La sémantique opérationnelle d'un opérateur est décrite par un algorithme implémentant la méta-opération correspondante. Dans cette section, ces algorithmes sont exprimés informellement en langage naturel. Leur implémentation en KerMeta [Triskell05], un langage de

méta-programmation permettant de décrire le comportement de méta-modèles, est présentée dans l'Annexe A.

II.2.1. Opérateurs de Recherche de Patrons de Procédé

Pour faciliter la lecture des algorithmes décrivant les opérateurs de recherche, nous montrons dans la Figure III-7 la partie du méta-modèle UML-PP concernant les concepts de patrons de procédé et leurs relations d'organisation.

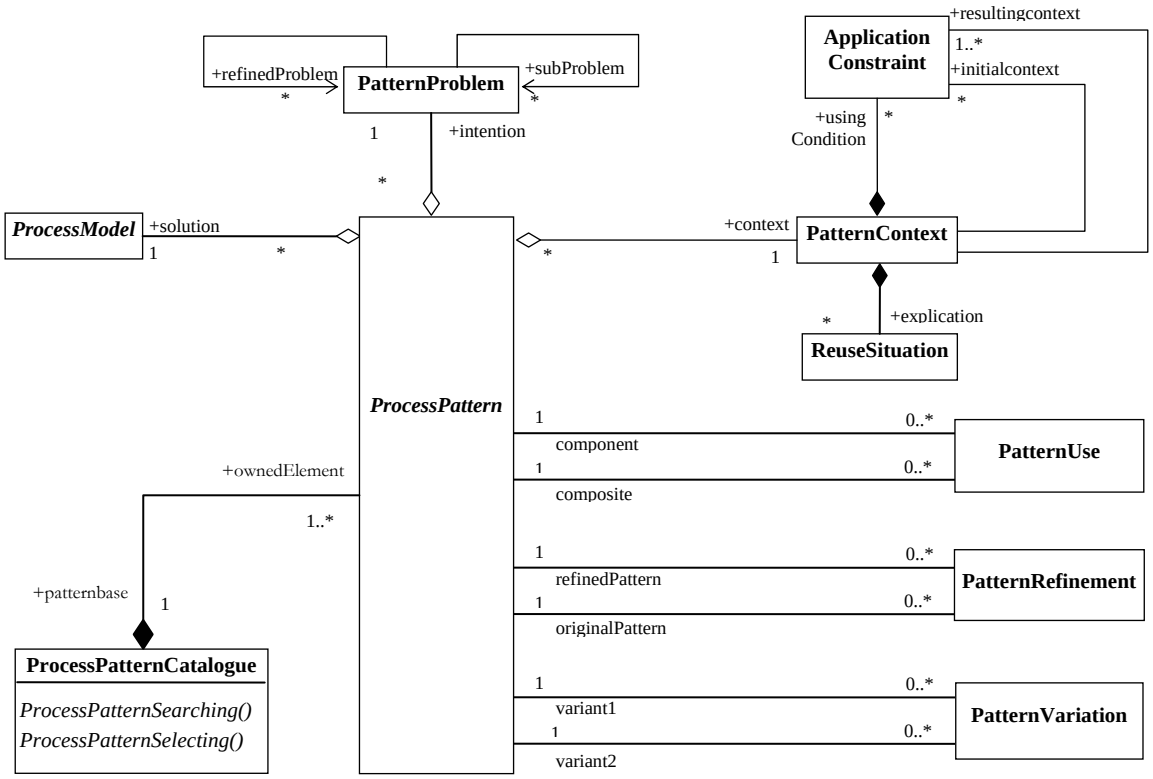


Figure III-7. Extrait du méta-modèle UML-PP décrivant l'organisation de patrons de procédé

Pour simplifier, dans les algorithmes suivants, on dénote :

- `PB = selft.ownedElement` : l'ensemble des patrons du catalogue examiné
- `pp ∈ PB` : un patron dans le catalogue des patrons examiné
- `pp.intention` : le problème adressé par le patron pp
- `refinedPP = pp.intention.refinedProblem` : l'ensemble des problèmes qui raffinent le problème du patron pp
- `pp.context` : le contexte spécifié pour le patron pp

II.2.1.1. ProcessPatternSearching

L'opérateur *ProcessPatternSearching* cherche dans un catalogue de patrons de procédé les patrons dont l'intention correspond au problème exprimé.

Soit *q* le problème en question, *res* l'ensemble des patrons choisis, *ProcessPatternSearching* réalise les actions suivantes :

Algorithme 1. Chercher des patrons convenables pour un problème donné

```
1. Pour chaque élément pp du catalogue PB
   1.1. Si l'intention du pp satisfait une des conditions suivantes :
          - q == pp.intention      //intention du pp identique à q
          - q ∈ refinedPP          //intention du pp plus abstraite que q
      1.1.a. res := res ∪{p}       //ajouter pp au res
2. Retourner res
```

Discussion

Pour simplifier la lecture, nous avons présenté l'algorithme de recherche le plus simple. On peut améliorer cet algorithme en considérant les relations organisationnelles entre les patrons du catalogue. Par exemple, lorsqu'un patron est trouvé, les patrons ayant des relations *PatternVariation* avec lui seront également ajoutés au résultat. Une telle approche pourrait aider à limiter le nombre de comparaisons à faire.

II.2.1.2. ProcessPatternSelecting

L'opérateur *ProcessPatternSelecting* sélectionne dans un ensemble de patrons de procédé les patrons les plus adaptés au contexte indiqué.

Soit *c* le contexte demandé, et *res* le patron sélectionné. *ProcessPatternSelecting* réalise les actions suivantes :

Algorithme 2. Sélectionner le patron le plus adapté à un contexte donné

```
1. var max : Integer                    //degré de conformité maximum actuel
2. PC := ProcessPatternSearching(q)     //PC est le sous-ensemble des patrons du PB
                                        //  satisfaisant le problème q
3. Pour chaque élément pp de l'ensemble PC
   3.1. Calculer son degré de conformité avec le contexte c :
      3.1.a. var conformdegree : Integer
      3.1.b. conformdegree := Conformity(pp,c)
   3.2.   Si (conformdegree > max)      //degré de conformité de pp
                                        //supérieur à la valeur maximum actuelle
      3.2.a.  max := conformdegree
      3.2.b.  res := pp                 //marquer pp comme un patron sélectionné
4. Retourner res
```

Discussion

Nous supposons ici que la fonction *Conformity(c1,c2)* est définie (c.f.II.1.1). Elle peut être exécutée automatiquement ou manuellement selon la complexité des problèmes et contextes exprimés dans les patrons.

S'il y a plusieurs patrons de même degré de conformité, cet algorithme ne retourne que le dernier trouvé. Une alternative serait de retourner tous les patrons adaptés et de laisser la décision finale au concepteur.

II.2.2. Opérateurs d'Imitation de Patrons de Procédé

L'automatisation des opérateurs d'imitation est importante pour garantir la correction des applications de patrons ainsi que pour réduire le temps de modélisation. Dans cette section nous présentons des algorithmes qui rendent une telle automatisation possible.

II.2.2.1. ProcessPatternBinding

La Figure III-8 montre un extrait du méta-modèle UML-PP décrivant la relation *ProcessPatternBinding*. La méta-opération *Unfold()* de la classe *ProcessPatternBinding* définit le comportement de cet opérateur.

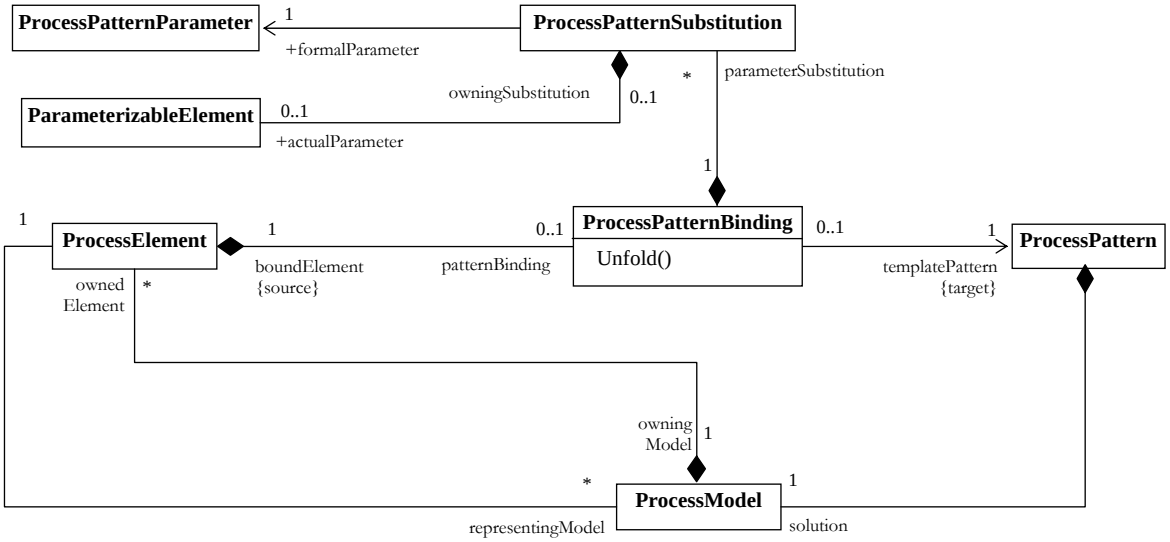


Figure III-8. Extrait du méta-modèle UML-PP décrivant la relation *ProcessPatternBinding*

L'opérateur *ProcessPatternBinding* génère un modèle décrivant un élément de procédé à partir de la solution d'un patron de procédé.

En exécutant l'opérateur *ProcessPatternBinding*, les scénarios suivants peuvent se produire :

- **Application exacte** : la relation *ProcessPatternBinding* n'a pas de paramètres, le modèle résultant est le duplicata de la solution du patron.
- **Application avec adaptation** : la relation *ProcessPatternBinding* est spécifiée avec des paramètres. Dans ce cas, le modèle résultant est élaboré en dupliquant la solution du patron et en substituant les éléments jouant le rôle de paramètre formel par des paramètres effectifs. La substitution doit prendre en compte l'existence des paramètres effectifs ainsi que leur niveau d'abstraction pour s'adapter à la spécification de l'élément à définir :

- Un paramètre effectif peut être un élément existant ou un élément n'appartenant pas au modèle conteneur et déclaré dans la substitution avec le type StringExpression.
- Un paramètre effectif peut être d'un niveau d'abstraction inférieur ou égal à celui de son paramètre formel.

Pour simplifier, dans l'algorithme suivant, on dénote :

- `pe = selft.boundElement` : l'élément à définir
- `pe.abslevel = pe.getAbstractionlevel()` : le niveau d'abstraction d'un élément
- `pe.type = pe.oclIsTypeOf()` : le type d'un élément (Task, Product,Role)
- `pm = pe.owningModel` : le modèle comportant *pe*
- `prm = pe.representingModel` : le modèle décrivant *pe*
- `pp = selft.templatePattern` : le patron à imiter
- `pp.abslevel = pp.getAbstractionlevel()` : le niveau d'abstraction d'un patron
- `subE = selft.parameterSubstitution` : l'ensemble des substitutions de paramètres
- `sub ∈ subE` : une substitution de paramètres dans l'ensemble des substitutions *subE*
- `sub.fp = sub.formalParameter` : le paramètre formel de la substitutions *sub*
- `sub.ap = sub.actualParameter` : le paramètre effectif de la substitutions *sub*

Soit *res* le modèle à élaborer pour décrire *pe*, *ProcessPatternBinding* réalise les actions suivantes :

Algorithme 3. Appliquer un patron pour générer un modèle décrivant un élément de procédé

```
1. res := pp.solution                    //Dupliquer la solution du patron pp et le nommer res

2. Si (subE ≠∅)                          // la relation ProcessPatternBinding est établie avec les
                                         // substitutions de paramètres,i.e. on substitue des
                                         //éléments de res représentant des paramètres formels par
                                         //des paramètres effectifs correspondants
   Pour chaque sub ∈ subE
   2.1. Si (sub.ap ∉ pm)                 //le paramètre effectif n'existe pas en dehors de subE,
                                         // il est simplement déclaré dans subE
      2.1.a. Si (pe.abslevel==pp.abslevel) //l'élément à définir pe est au même niveau
                                           //d'abstraction que le patron à imiter pp
         2.1.a.i.  Selectionner e ∈ res | e==sub.fp
         2.1.a.ii. e := sub.ap             //renommer l'élément représentant paramètre
                                           //formel du sub par le paramètre effectif
      2.1.b. Si (pe.abslevel<pp.abslevel)  //l'élément à définir pe est au même niveau
                                           //d'abstraction ou à un degré inférieur à
                                           //celui du patron à imiter pp
         var realap:ProcessElement
```

```
        2.1.b.i.  realap:= CreateElement(pe.abslevel,sub.fp.type)
                                  // Créer le paramètre effectif réel ayant le niveau
                                  //d'abstraction de pe et le type du paramètre formel
        2.1.b.ii. realap.nom:= sub.ap
        2.1.b.iii. Selectionner e ∈ res | e==sub.fp
        2.1.b.iv. e := realap     //substituer l'élément représentant le paramètre
                                  //formel de sub par le paramètre effectif créé
   2.2. Si (sub.ap ∈ pm)          //le paramètre effectif existe déjà
        var refap : Reference of ProcessElement
        2.2.a.i. Selectionner oe ∈ pm |oe==sub.ap
                                  //identifier l'élément d'origine désigné en tant
                                  //que paramètre effectif
        2.2.b. refap := Import(oe)   //importer l'élément d'origine du paramètre effectif
        2.2.c. Selectionner e ∈ res | e==sub.fp
        2.2.d. e:=refap           //substituer le paramètre formel par l'élément importé
3. prm :=res                      //créer le lien de représentation entre res et pe
```

Remarque

Nous supposons ici que l'opération *CreateElement(abs,t)* est définie pour créer un élément de procédé de type *t* au niveau d'abstraction *abs*. Nous supposons également que l'opération *Import(e)* est définie pour créer un élément qui est une référence à l'élément *e*.

II.2.2.2. ProcessPatternApplying

La Figure III-9 montre un extrait du méta-modèle UML-PP décrivant la relation *ProcessPatternApplying*. La méta-opération *Unfold()* de la classe *ProcessPatternApplying* définit le comportement de cet opérateur.

L'opérateur *ProcessPatternApplying* applique la solution d'un patron de procédé à un modèle de procédé pour l'enrichir ou le (re)structurer.

Un opérateur *ProcessPatternApplying* doit être spécifié avec des règles de substitution de paramètres (c.f. la section II.3.1.3 du Chapitre II). Nous distinguons donc les scénarios d'exécution possibles suivants :

- **Application en mode Change** : dans ce cas, le modèle résultant est le duplicata de la solution du patron avec les éléments représentant les paramètres formels remplacés par des éléments du modèle cible représentant les paramètres effectifs.
- **Application en mode Replace ou Extend** : dans ces deux cas, le modèle résultant est la fusion du duplicata de la solution du patron avec le modèle cible. La fusion est réalisée en réunissant les éléments des deux modèle et en remplaçant des éléments représentant des paramètres formels par des éléments représentant des paramètres effectifs. Une telle fusion peut conduire à des incohérences dans le modèle résultant. Par conséquent, il faut réaliser un traitement des conflits lors de la fusion.

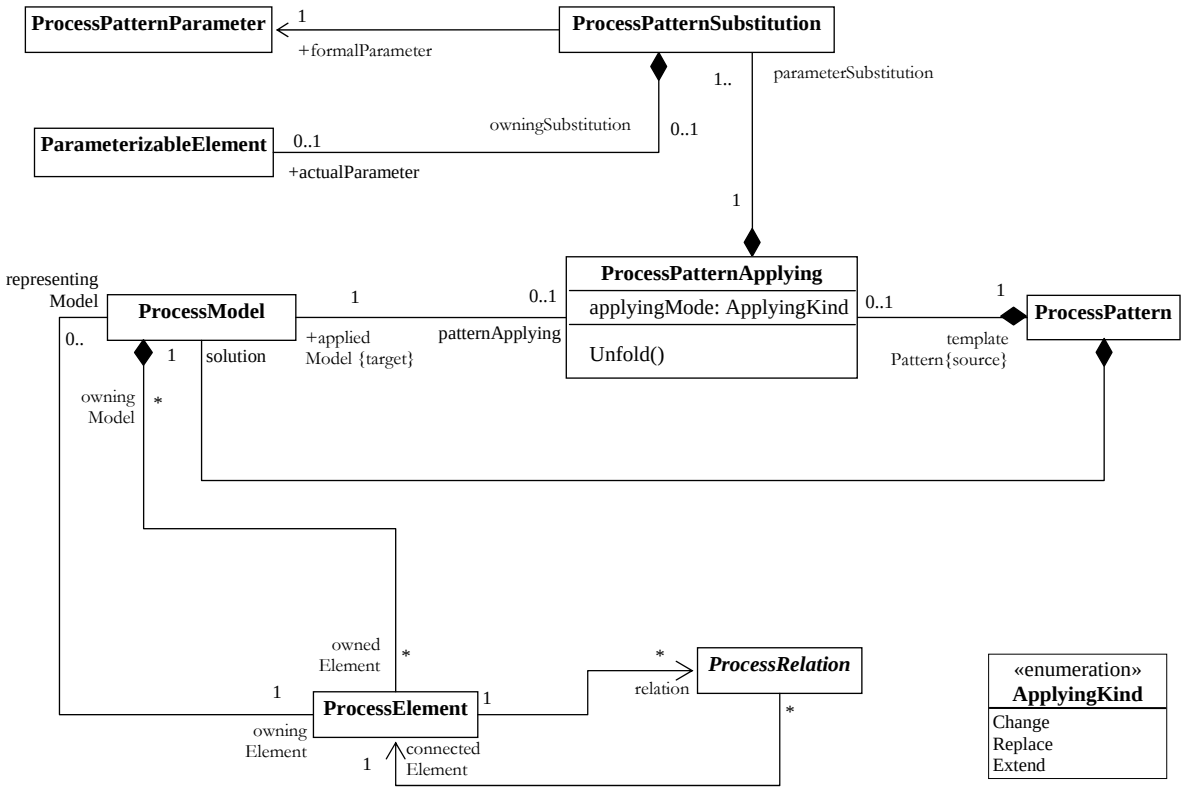


Figure III-9. Extrait du méta-modèle UML-PP décrivant la relation *ProcessPatternApplying*

Pour simplifier, dans l'algorithme suivant, on dénote :

- `pm = selft.appliedModel` : *le* modèle à enrichir ou restructurer
- `pe.relations` : l'ensemble des relations dont *pe* est l'élément source
- `r` : une relation de procédé
- `r.type :` le type de la relation *r*
- `r.source :` l'élément source de la relation *r*
- `r.dest :` l'élément cible de la relation *r*
- `pp = selft.templatePattern` : le patron à imiter
- `subE = selft.parameterSubstitution` : l'ensemble des substitutions de paramètres
- `sub ∈ subE` : une substitution de paramètres dans l'ensemble des substitutions *subE*
- `sub.fp = sub.formalParameter` : paramètre formel de la substitutions *sub*
- `sub.ap = sub.actualParameter` : paramètre effectif de la substitutions *sub*
- `applMode= self.applyingMode` : mode d'application de la relation *ProcessPatternApplying*

Les préconditions suivantes doivent être satisfaites pour appliquer l'opérateur *ProcessPatternApplying :*

- `subE ≠ ∅ //`
- `∀ sub ∈ subE, (sub.fp∈pp.solution) ∧ (sub.ap∈pm)`

Soit *res* le modèle résultant en remplacement de *pm*, *ProcessPatternApplying* réalise les actions suivantes pour produire *res* en fusionnant *pm* avec la solution de *pp :*

Algorithme 4. Appliquer un patron pour restructurer ou enrichir un modèle de procédé

1. `res := pp.solution` *//Dupliquer la solution du patron pp et le nommer res*

2. *//Substituer les éléments de res représentant des paramètres formels par les éléments de pm //représentant des paramètres effectifs :*

Pour chaque paire sub ∈ subE

2.1. Sélectionner e ∈ res | e == sub.pf

2.2. e := sub.ap ∈ pm

3. Si applMode == *Change* *//le modèle résultant est élaboré en utilisant le duplicata de //la solution du patron avec des éléments remplacés par ceux //du modèle cible*

aller étape 6

4. *//ajouter à res les éléments de pm qui ne sont pas spécifiés comme paramètres effectifs*

Pour chaque e ∈ pm | e ∉ {sub.ap}

4.1. res :=res ∪ {e}

5. *//Travailler sur le modèle res pour rétablir les relations entre les éléments originels de pm, et //éventuellement résoudre les conflits avec les relations imposées par les éléments originels //de la solution du patron pp :*

Pour chaque élément e ∈ *pm* :

5.1. var RE : Set of ProcessRelation

5.2. RE :=e.relations *//identifier l'ensemble des relations de pm dont e est l'élément source*

5.3. Pour chaque r ∈ RE *//e ∈ pm est l'élément source de // la relation r*

var f,eres,fres : ProcessElement

5.3.a. f := r.connectedElement *//f ∈ pm est l'élément cible de //la relation r*

5.3.b. *//identifier eres, fres ∈ res correspondant à e,f ∈ pm*

Selectionner eres ∈ res | eres ==e

Selectionner fres ∈ res | fres ==f

5.3.c. *//S'il n'y a pas une même relation r entre eres et fres*

Si ∀ rres ∈ eres.relation |rres.type == r.type,

(fres ∉ rres.connectedElement)

//établir une relation de même type que r entre eres et fres

var newr :ProcessRelation

newr.source :=eres

newr.dest :=fres

eres.relation := eres.relation ∪ {newr}

5.3.d. *//Sinon, l'ajout de r dans le modèle résultant peut conduire à des conflits*

5.3.d.i. *//S'il y a une relation rres entre eres et fres qui est la relation //inverse de r : conflit direct*

Si ∃ rres ∈ fres.relation |rres.type == r.type, (eres ∈ rres.connectedElement)

a. Si applMode == *Extend* *//privilégier les relations de pm*

// supprimer la relation rres entre eres et eres

fres.relation:=fres.relation \{rres}

//établir une relation de même type que r entre eres et fres

var newr :ProcessRelation

newr.source :=eres

newr.dest :=fres

eres.relation := eres.relation ∪ {newr}

b. Si applMode == *Replace* *//privilégier les relations de pp*

//conserver la relation z entre eres et fres

aller étape 6

5.3.d.ii. Si l'ajout de *r* peut conduire à des incohérences dans *res* : conflit indirect

Identifier et traiter les conflits potentiels (c.f. Discussion ci-dessous)

ConflictsResolving()

6. pe := pm.owningElement *//l'élément pe est décrit par pm*

7. *//remplace pm par le modèle résultant res*

pe.representingModel := res

Discussion

Pour résoudre les conflits indirects, l'opération *ConflictsResolving()* est utilisée. Nous discutons ci-dessous des conflits possibles à régler en appliquant un patron de procédé pour modifier un modèle de procédé.

Dans le cadre de cette thèse, nous supposons que les modèles de procédé capturés dans les patrons de procédé et les modèles à modifier sont décrits dans un même LDP. Cela évite les conflits concernant la définition d'éléments. Nous supposons également qu'il n'y a pas de conflits de nom ou de caractéristiques d'éléments. Ainsi, nous pouvons nous concentrer sur des conflits concernant des relations entre éléments d'un modèle.

La Figure III-10 montre un extrait du méta-modèle décrivant les différentes relations possibles entre éléments de procédé.

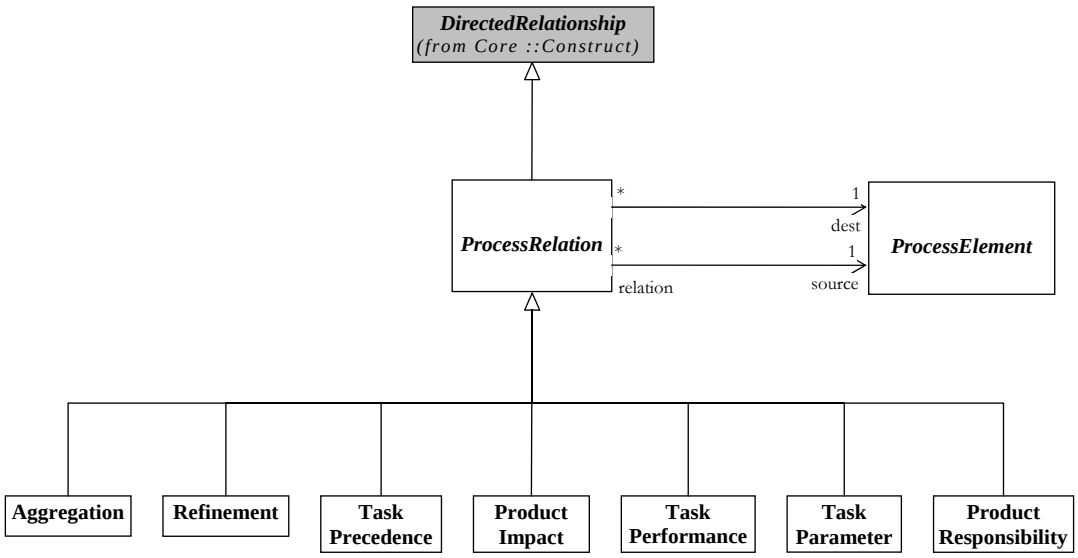


Figure III-10. Relations définies entre éléments de procédé

Dans notre méta-modèle, une relation de procédé (*ProcessRelation*) est une relation binaire qui relie un élément de procédé *source* à un élément de procédé cible (*dest*). Nous distinguons deux types de relations de procédé :

- **Relation homogène :** la source et la cible de la relation sont du même type d'élément de procédé.

Les relations *Aggregation*, *Refinement*, *TaskPrecedence* et *ProductImpact* sont des relations homogènes.

Une relation homogène est irréflexive, antisymétrique et transitive. En établissant un lien de type relation homogène entre deux éléments de procédé, il faut vérifier les propriétés liées à la réflexivité et la symétrie de la relation. Autrement dit, lorsque l'établissement d'un lien entre deux éléments du modèle conduit à l'apparition d'une relation cyclique, un conflit se produit.

Par conséquent, en ajoutant un nouveau lien *r* entre *source* et *dest* - ici *r* est une relation originelle du modèle cible de la relation *ProcessPatternApplying* - il faut vérifier la condition suivante pour assurer la cohérence d'un modèle dans ce cas :

(1) Dans le contexte d'un élément source qui est relié à r:

$$\{source\} \cap \text{fermeture transitive } (source.relation.dest) = \varnothing$$

Parmi les relations homogènes, la relation *Refinement* demande une autre vérification pour assurer la contrainte sur le nombre d'éléments originels d'un raffinement (c.f. section II.1.2, contrainte [C5]) :

(2) Dans le contexte d'un élément source relié à r, r étant une relation de type Refinement :

$$size(\{source.relation.dest\}) = 1$$

S'il y a un conflit, le traitement dépend du mode d'application choisi :

- **applyingMode = Replace** : supprimer *r*.
- **applyingMode = Extend** : garder *r* et supprimer une des relations dans la fermeture transitive de *r* pour éliminer le cycle. Cependant, le choix de la relation à enlever n'est pas évident.

Nous illustrons dans la Figure III-11 ci-dessous quelques exemples de conflits se produisant dans le contexte de relations homogènes lors de l'application d'un patron de procédé pour restructurer un modèle de procédé.

La Figure III-11 (a) montre le modèle cible à restructurer, tandis que la Figure III-11 (b) montre le patron source utilisé pour restructurer le modèle cible et la relation *ProcessPatternApplying* définie entre eux selon le mode *extend.*

La Figure III-11 (c) montre le modèle résultant élaboré par imitation du patron et substitution des paramètres. À cette étape, la structure du modèle résultant est celle du patron, mais les éléments représentant les paramètres formels sont remplacés par les paramètres effectifs (par exemple *X* est remplacé par *A*, *Y* est remplacé par *B*, etc.)

La Figure III-11 (d) montre le modèle résultant après le rétablissement des relations définies entre les éléments originels du modèle cible. Ces relations sont représentées en gras avec les noms soulignés. On peut voir que l'ajout de ces relations conduit aux conflits suivants :

- Conflits de relation de R*efinement* : les relations de raffinement entre *A*, *M* et *G* forment un cycle, elles violent donc la contrainte (1) ; il y a deux relations de raffinement dont C est l'élément source (l'une entre *A* et *C,* l'autre entre *C* et *B*), ce qui viole la contrainte (2).
- Conflits de relation *ProductImpact* : entre *B* et *C* il y a deux relations d'impact inverses, l'une spécifiant que *B* dépend de *C*, l'autre spécifiant que *C* dépend de *B*. Ceci viole la contrainte (1).
- Conflits de relation d'*Aggregation* : entre *C* et *D* il y a deux relations d'agrégation inverses, l'une venant du modèle cible en spécifiant que *D* est un composant de *C,* l'autre venant du patron source en spécifiant que *C* est un composant de *D*. Cela viole aussi la contrainte (1).

Pour régler ces conflits, il faut supprimer certaines relations. Comme le mode d'application choisi est «Extend», les relations originelles du modèle cible sont à conserver et les relations du patron qui sont en contradiction avec ces relations originelles sont à enlever (Figure III-11 (e)). Comme nous l'avons remarqué, le choix de la relation à enlever n'est pas évident s'il s'agit d'un cycle. Par exemple pour éviter le cycle de relations de raffinement entre *A*, *M* et *G*, on peut supprimer la relation entre *A* et *M* (c'est le cas dans la Figure III-11 (e)), ou la relation entre *M* et *G*.

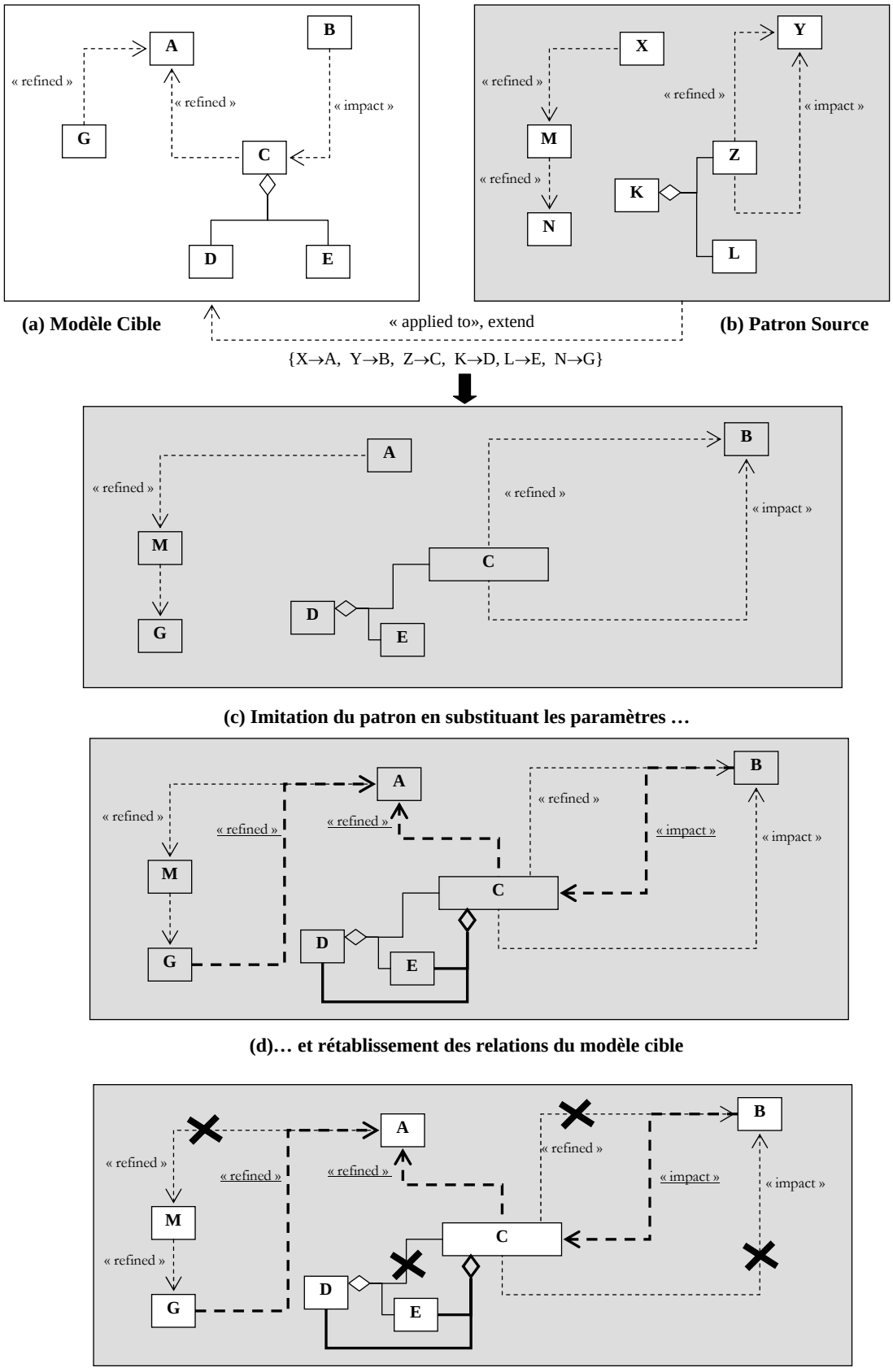


Figure III-11. Exemple de gestion de conflits de relations homogènes

- **Relation hétérogène :** dans ce type de relation, la source et la cible d'une relation sont des éléments de différents types d'élément de procédé.

Les relations *TaskParameter*, *TaskPerformance*, et *ProductResponsibility* sont des relations hétérogènes.

Une relation hétérogène est irréflexive et antisymétrique mais pas transitive. Par conséquent, en établissant un lien de type relation hétérogène entre deux éléments de procédé, il n'est pas nécessaire de vérifier les propriétés liées à la réflexivité et à la symétrie de la relation. Autrement dit, l'établissement d'un tel lien entre deux éléments du modèle ne conduit pas à un conflit sous forme de relation cyclique.

Cependant, l'ajout d'une relation hétérogène peut avoir des impacts sur les autres relations et conduire à des incohérences ou des redondances dans le modèle résultant. De tels conflits sont très compliqués et dépendent de chaque type de relation.

Dans la suite nous identifions certains problèmes potentiels concernant l'interaction entre relations hétérogènes, mais une analyse complète des conflits de ce type demande des investigations complémentaires qui sont en cours d'étude.

(i) Conflits liés à la relation TaskParameter

Un conflit potentiel lié à l'ajout d'une relation *TaskParameter* est l'incohérence entre les produits entrées/sorties et les dépendances *TaskPrecedence* entre deux tâches.

Nous montrons dans la Figure III-12 un exemple d'application de patron en mode *extend.*

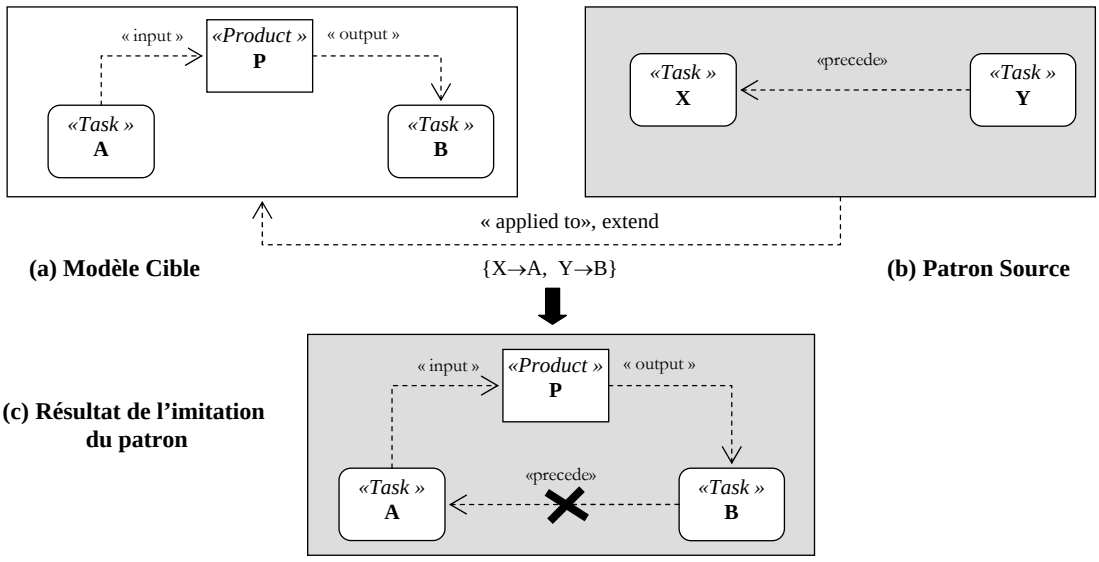


Figure III-12. Exemple de conflits liés aux relations *TaskParameter* et *TaksPrecedence*

D'après la contrainte [C30] de la section II.1.5 du Chapitre II, s'il existe une dépendance *TaskPrecedence* entre deux tâches, les produits de sortie du successeur ne peuvent pas être des produits d'entrée du prédécesseur. Cette contrainte est violée en fusionnant les deux modèles de

l'exemple ci-dessus. Pour régler le conflit, la relation *TaskPrecedence* définie dans le patron source est supprimée car le mode d'application est «Extend».

(ii) Conflits liés à la relation TaskPerformance

Voici des conflits potentiels liés à l'ajout d'une relation *TaskPerformance* :

- **Situation 1** : un rôle réalise et assiste une même tâche. La Figure III-13 montre un exemple de cette situation.

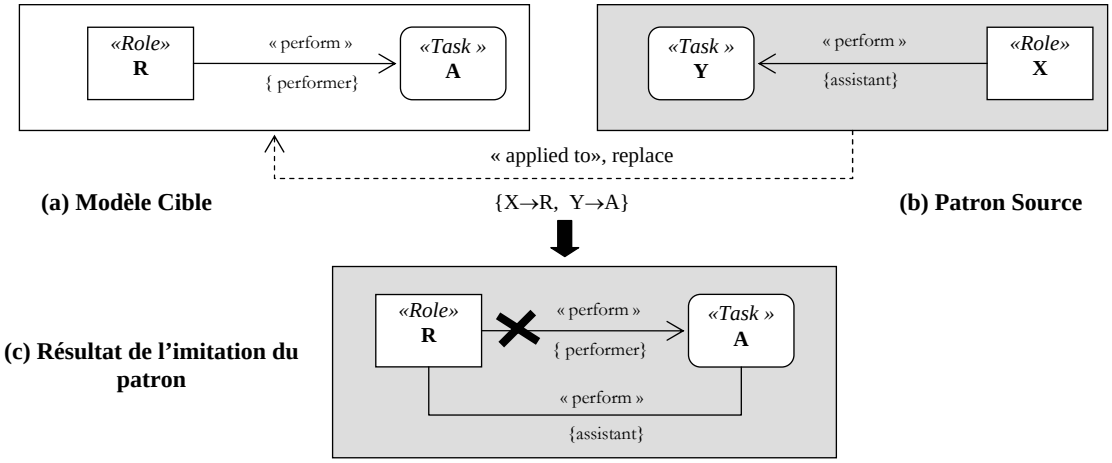


Figure III-13. Exemple 1 de conflits liés à la relation *TaskPerformance*

D'après la contrainte [C17] de la section II.1.4 du Chapitre II, il ne peut pas y avoir plus d'une relation *TaskPerformance* entre un rôle et une même tâche. Cette contrainte est violée en fusionnant les deux modèles de l'exemple ci-dessus. Pour régler le conflit, la relation *TaskPerformance*{*performer*} définie dans le modèle cible est supprimée car le mode d'application est «Replace».

- **Situation 2** : une tâche est réalisée par deux rôles. La Figure III-14 montre un exemple de cette situation.

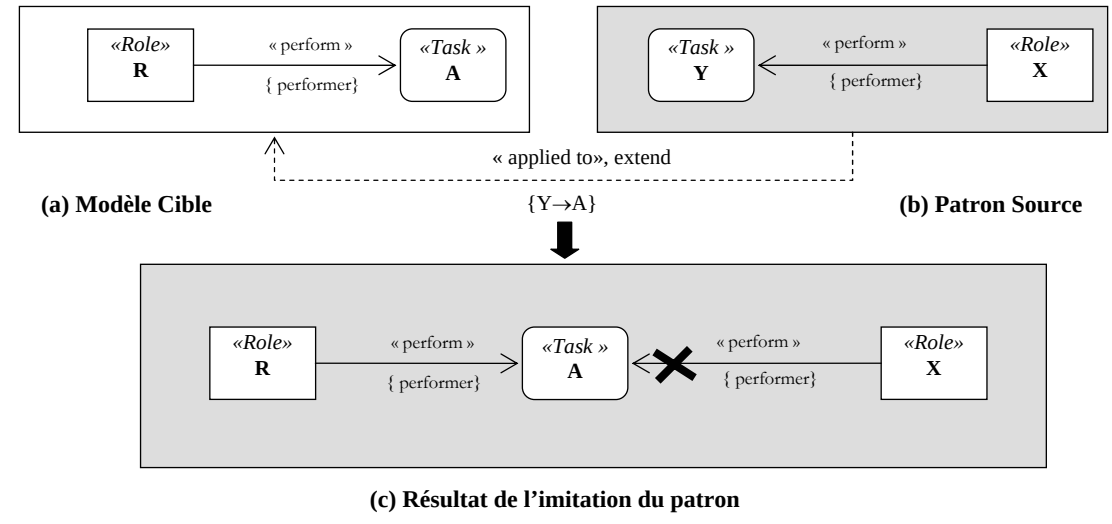


Figure III-14. Exemple 2 de conflits liés à la relation *TaskPerformance*

D'après la contrainte [C24] de la section II.1.5 du Chapitre II, une tâche est réalisée par un seul rôle. Cette contrainte est violée en fusionnant les deux modèles de l'exemple ci-dessus. Pour régler le conflit, la relation *TaskPerformance*{*performer*} définie dans le patron source doit être supprimée car le mode d'application est «Extend».

(iii) Conflits liés à la relation ProductResponsability

L'ajout d'une relation *ProductResponsability* peut conduire à des redondances ou à des incohérences :

- **Situation 1** : un rôle est en même temps responsable d'un produit et d'un de ses composants. La Figure III-15 montre un exemple de cette situation.

Dans cet exemple, après la fusion de deux modèles, il n'y a pas de conflit. Cependant, comme il y a deux relations *ProductResponsability* entre le rôle R et les produits A et M, il peut y avoir une incohérence (redondance) car on ne sait pas si R est responsable d'un seul composant M ou de tout le produit A.

Pour régler l'incohérence dans cet exemple, la relation *ProductResponsability* entre R et A définie dans le modèle cible est supprimée car le mode d'application est «Replace».

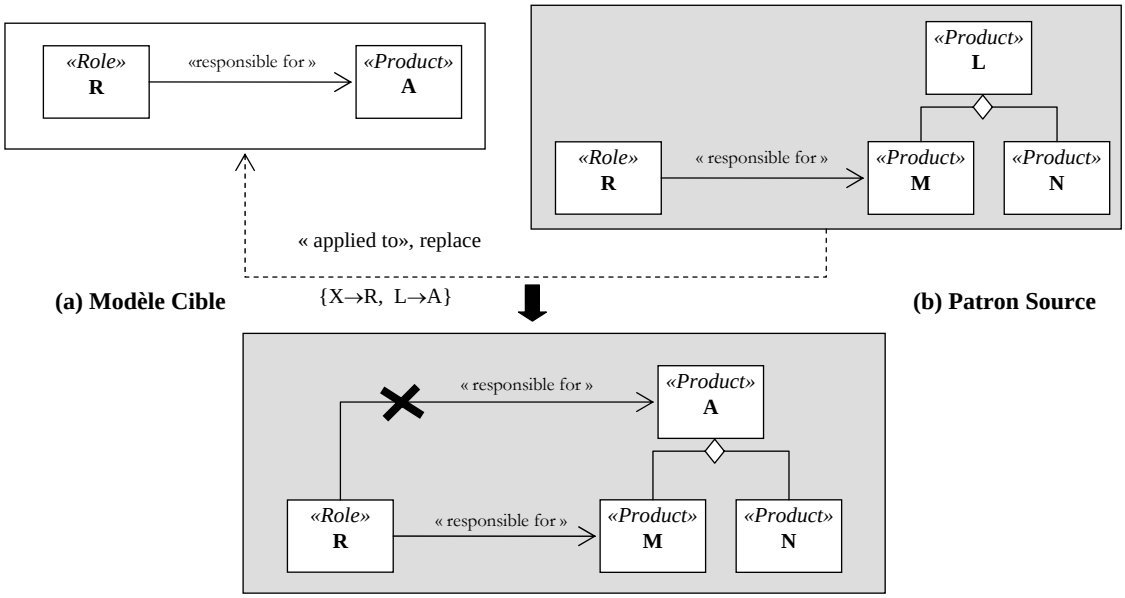


Figure III-15. Exemple 1 de conflits liés à la relation *ProductResponsability*

- **Situation 2** : un rôle est responsable d'un produit mais il ne réalise pas la tâche qui crée ou manipule ce produit. La Figure III-16 montre un exemple de cette situation.

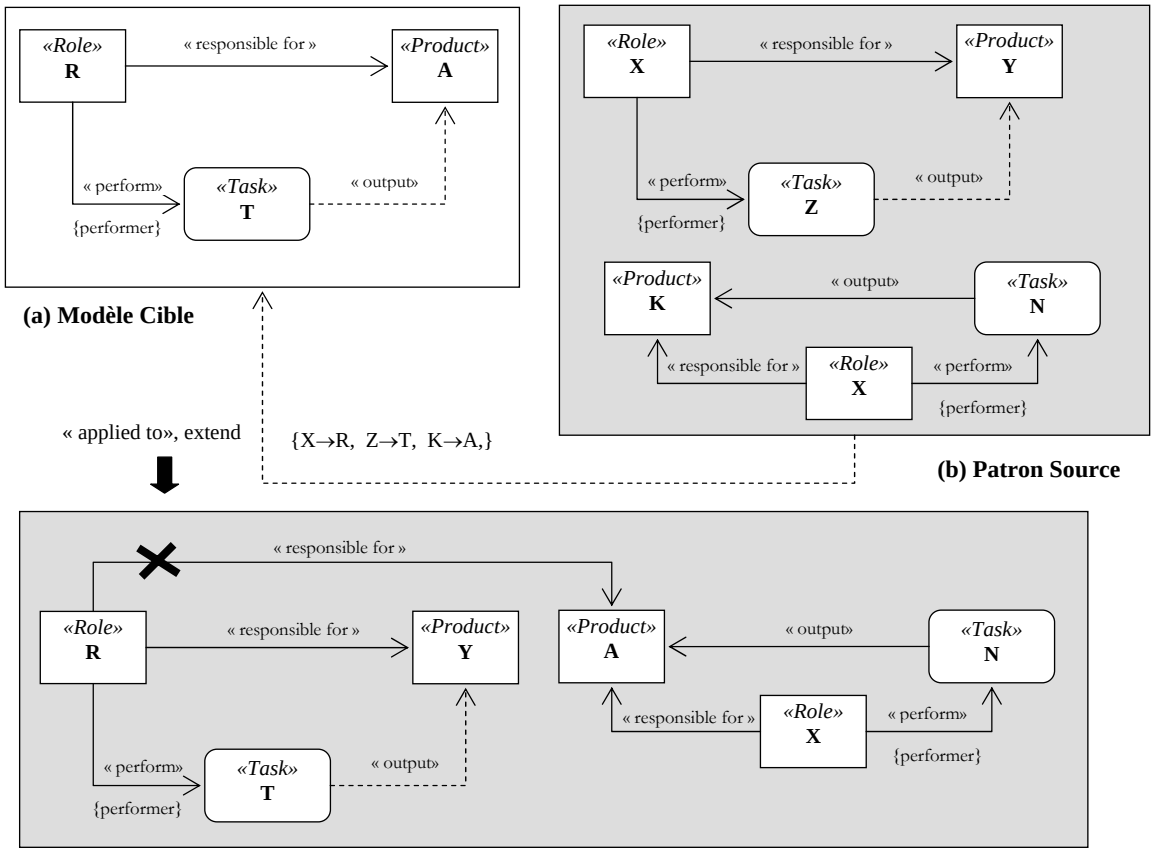


Figure III-16. Exemple 2 de conflits liés à la relation *ProductResponsability*

Dans cet exemple, le modèle cible et le patron source sont cohérents avant la fusion. Mais après la fusion, une incohérence se produit : le rôle R est encore responsable du produit A alors qu'il réalise la tâche T qui maintenant crée le produit Y. Par contre, R ne réalise pas la tâche N qui crée le produit A. Ceci viole la contrainte [C19] (c.f. section II.1.4 du Chapitre II) qui demande que le responsable d'un produit réalise la tâche manipulant ce produit. Une solution pour régler ce conflit est de supprimer la relation entre R et A définie dans le modèle cible.

II.2.3. Synthèse

Nous avons défini dans cette section des opérateurs de réutilisation de patrons de procédé. Ces opérateurs fournissent des moyens pour manipuler les modèles de procédé et les patrons de procédé. Nous avons donné une sémantique opérationnelle à ces opérateurs pour les rendre automatisables. Lors de l'application de l'opérateur *ProcessPatternApplying*, des conflits structurels peuvent apparaître. Nous avons identifié ces conflits potentiels et proposé des moyens de résolution systématiques dans la discussion précédente. Cependant, pour être optimale, la résolution de ces conflits nécessite l'intervention du concepteur de procédé.

Pour compléter la méthode de modélisation à base de patrons réutilisables, nous proposons dans la section suivante une démarche pour guider les concepteurs dans l'application des opérateurs définis ci-dessus.

III. LE MÉTA-PROCÉDÉ PATPRO

Il y a une analogie entre le développement de logiciels et le développement de procédés car les procédés logiciels eux-mêmes peuvent être considérés comme des logiciels [Osterweil87]. La modélisation de procédés est en fait un développement où les produits sont des modèles de procédé.

Toute méthode de développement comporte une démarche à suivre pour atteindre des objectifs donnés. Une méthode de modélisation par réutilisation de patrons réutilisables ne déroge pas à ce principe. La démarche fournie doit permettre au concepteur d'effectuer un suivi pas-à-pas des étapes de modélisation de procédés en favorisant la réutilisation de patrons de procédé. La définition d'une telle démarche doit donc consister non seulement en la description des activités de modélisation, mais aussi en leur ordonnancement et en l'expression des modèles et artefacts requis et produits. Autrement dit, la démarche proposée doit être représentée sous forme d'un méta-procédé à grain fin.

Notre objectif est de fournir un méta-procédé à grain fin, nommé PATPRO, pour la construction ou l'amélioration de modèles de procédé, en guidant les concepteurs dans la réutilisation de patrons de procédé.

Nous n'introduisons pas une perspective méthodologique complètement nouvelle pour la modélisation de procédés, mais raffinons les approches existantes en mettant l'accent sur la réutilisation de connaissances de procédé. La démarche que nous proposons est inspirée des méthodes de modélisation de procédés telles que PRIM [Madhavji90], [Gomaa00] et [Hollenbach96]. Elle s'appuie également sur des normes d'évaluation et d'amélioration de procédés comme CMMI [CMMI02] et ISO15504 [ISO98].

Dans le cadre de cette thèse, nous nous sommes centrés sur les étapes d'analyse et de conception de modèle de procédé. Plus précisément, nous décrivons les méta-tâches de ces étapes pour construire statiquement les modèles de procédé. Nous supportons le raffinage de modèles pendant la modélisation, mais nous ne prenons pas en compte le problème de l'adaptation dynamique des modèles de procédé[1]. Pour représenter le méta-procédé PATPRO, nous utilisons le langage de description de procédés défini par le méta-modèle UML-PP présenté dans le Chapitre II.

Dans la suite, nous décrivons d'abord succinctement la structure générale de PATPRO (section III.1). Les rôles et les méta-tâches du PATPRO sont ensuite présentés en détail dans les sections III.2 et III.3.

III.1. DESCRIPTION GÉNÉRALE DU MÉTA-PROCÉDÉ PATPRO

Par analogie avec le développement du logiciel et en tenant en compte des spécificités des procédés, le méta-procédé PATPRO (Figure III-17) se présente sous la forme des quatre

[1] Un modèle de procédé peut comporter des informations incomplètes qui seront raffinées ou complétées pendant la modélisation pour tenir compte du contexte spécifique d'un projet. Dans notre méta-procédé, ce raffinage est effectué dans la phase de simulation/exécution au moment d'instancier le modèle de procédé, mais avant l'exécution de l'instance.

méta-tâches suivantes :

- **Analyser les Besoins** : cette méta-tâche analyse les caractéristiques du procédé à modéliser, et identifie les besoins de modélisation.
- **Modéliser un Procédé** : cette méta-tâche élabore le modèle d'un procédé à partir de sa spécification. C'est la tâche centrale de la modélisation. Nous utilisons l'approche centrée-activité[1] pour construire les procédés.
- **Simuler un Procédé** : cette méta-tâche simule l'exécution d'un modèle de procédé par l'intermédiaire d'un moteur d'exécution de procédés.
- **Évaluer un Procédé** : cette méta-tâche évalue un modèle de procédé en le comparant avec des critères prédéfinis.

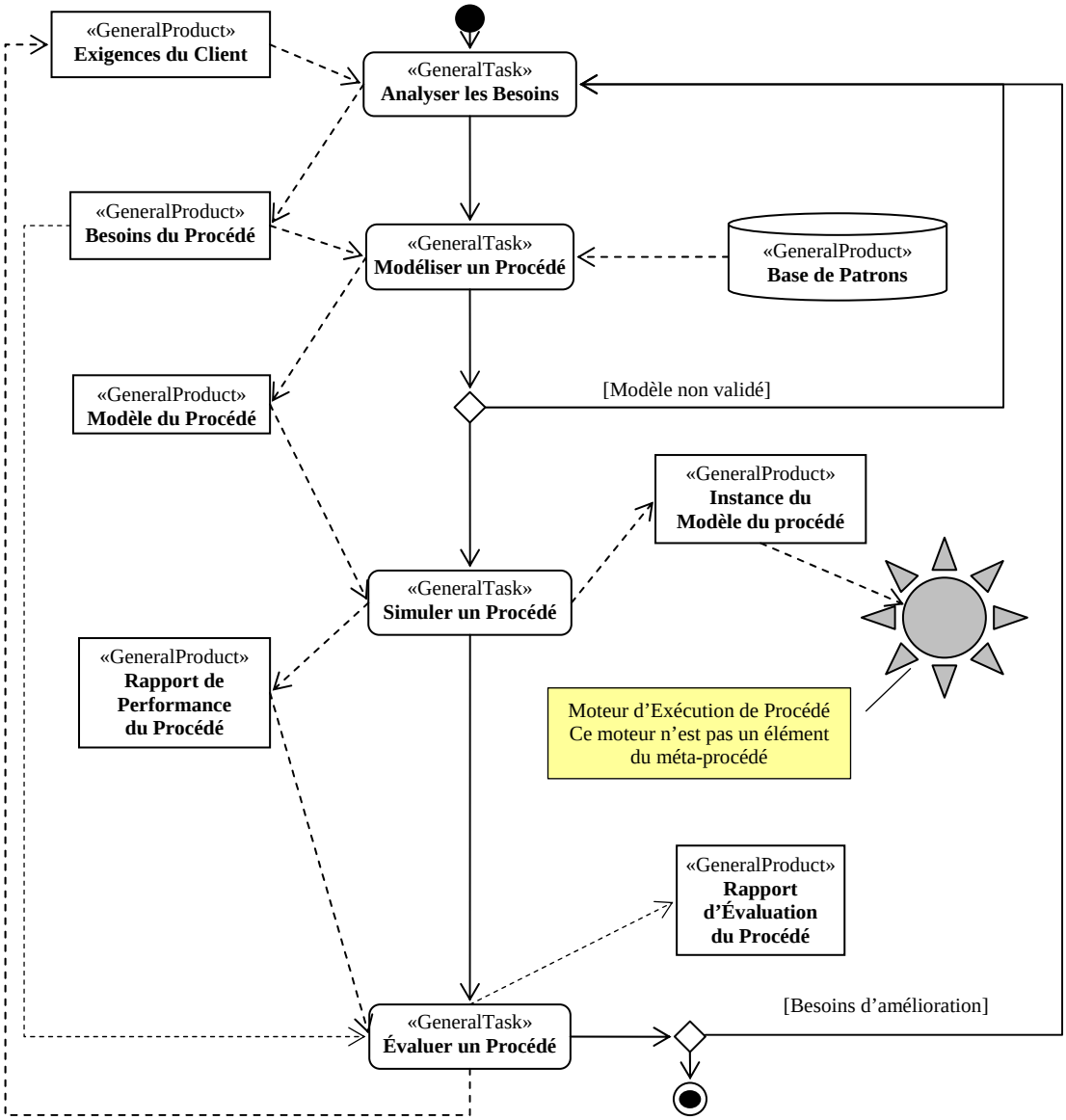


Figure III-17. Cycle de vie global du méta-procédé PATPRO

[1] Cette approche voit un procédé comme un ensemble d'activités ordonnées. L'activité est donc l'élément central de procédé, les autres éléments ayant pour but de compléter la description des activités.

Comme pour le développement de logiciels, il y a plusieurs façons d'organiser les méta-tâches selon différents modèles de cycle de vie. En remarquant que les modèles de procédé logiciel sont peu évolutifs durant leur conception[1], nous choisissons un cycle de vie inspiré du modèle de la cascade pour organiser les méta-tâches du PATPRO.

Nous décrivons le méta-procédé PATPRO à un niveau d'abstraction général pour permettre sa spécialisation plus tard dans des projets de modélisation spécifiques. Les éléments de PATPRO sont donc modélisés par des *GeneralElement* du méta-modèle UML-PP.

Un cycle de modélisation de procédés commence avec la tâche *Analyser les Besoins* pour produire une spécification de *Besoins du Procédé*. Cette spécification est ensuite utilisée par la tâche *Modéliser un Procédé* pour élaborer le *Modèle du procédé* en réutilisant éventuellement des patrons d'une *Base de Patrons de procédé*. Le modèle créé peut être instancié par la tâche *Simuler un Procédé* et l'instance du modèle exécutée par un *Moteur d'exécution de procédé* [2]. Après la simulation, un *Rapport de Performance du procédé* est produit et fourni à la tâche *Évaluer un Procédé* pour identifier éventuellement de nouveaux besoins. Un autre cycle de modélisation peut être requis selon de nouvelles exigences issues du *Rapport d'Évaluation valuer du Procédé*.

III.2. LES RÔLES DU MÉTA-PROCÉDÉ

Nous définissons trois rôles principaux pour représenter les responsabilités des participants du méta-procédé :

- **Analyste de Procédé** : ce rôle est responsable des activités concernant l'analyse des besoins de procédé et l'évaluation des procédés. En général, un acteur qui joue ce rôle doit maîtriser certaines techniques d'analyse de besoins (par dialogue, par scénarios, etc.) et des méthodes d'évaluation de procédé (standards, des métriques, etc.).
- **Concepteur de Procédé** : ce rôle a pour objectif de réaliser des activités concernant l'élaboration des modèles de procédé. La capacité d'utiliser certains LDP et éventuellement des outils de modélisation de procédés est requise pour ce rôle.
- **Opérateur de procédé** : ce rôle a pour but d'exécuter une instance d'un modèle de procédé. Autrement dit, il met en oeuvre un processus défini par le modèle de procédé. La compétence basique attendue de ce rôle est la connaissance de la gestion de projet (méthodes, outils, etc.).

Ces rôles peuvent être raffinés, spécialisés selon les besoins et les compétences spécifiques d'une équipe de développement de procédés concrets. Nous montrons dans la Figure III-18 le modèle de rôles de PATPRO et un exemple de raffinement.

[1] En général, les nouveaux besoins de modélisation d'un procédé (pour l'adapter, l'améliorer) n'apparaissent qu'après un cycle de vie entier du méta-procédé (afin de bénéficier de retours d'expérience et d'analyses de performance).

[2] En supposant que le modèle élaboré soit décrit en un LDP formel, et qu'il existe un moteur d'exécution pour ce LDP.

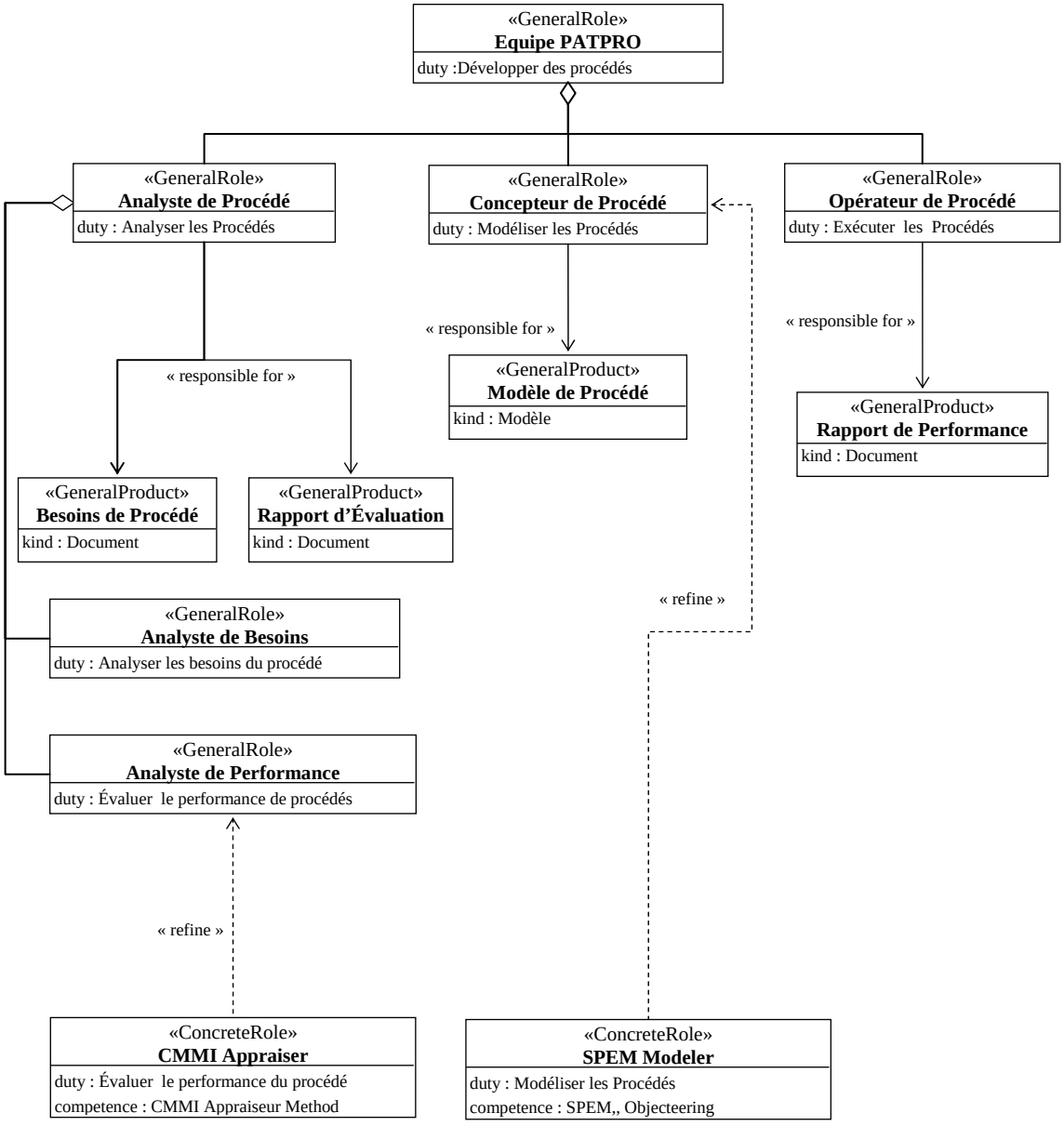


Figure III-18. Les rôles du méta-procédé PATPRO

III.3. LES MÉTA-TÂCHES DU MÉTA-PROCÉDÉ

Dans la suite, nous utilisons une présentation en quatre rubriques pour décrire une tâche[1] :

- Objectif : cette rubrique spécifie le but de la tâche.
- Agents : cette rubrique décrit les intervenants de la tâche
- Produits : cette rubrique spécifie les artéfacts consommés ou fabriqués par la tâche.
- Activités : cette rubrique décrit la structure et le comportement de la tâche, plus précisément la décomposition de la tâche en sous-tâches et l'ordonnancement de ces sous-tâches.

[1] Une tâche présentée dans le contexte du méta-procédé est considérée comme une méta-tâche.

III.3.1. Analyser les Besoins

Objectif

Cette tâche concerne la spécification du procédé à modéliser. Elle consiste à définir les caractéristiques du procédé, les objectifs et les besoins attendus de la modélisation et de l'exécution du procédé.

Agents

- Responsable : *Analyste de Procédé*
- Participant : clients (experts du domaine, équipe de développement)

Produits

- Entrée : *Exigences du Client*
- Sortie : *Besoins du Procédé*

A partir du document d'*Exigences du Client* fourni par le client, l'analyste de procédé va élaborer le document de *Besoins du Procédé* comprenant deux sous-produits : *Spécification du Procédé,* et *Objectifs et Contraintes de Modélisation et d'Exécution* (Figure III-19).

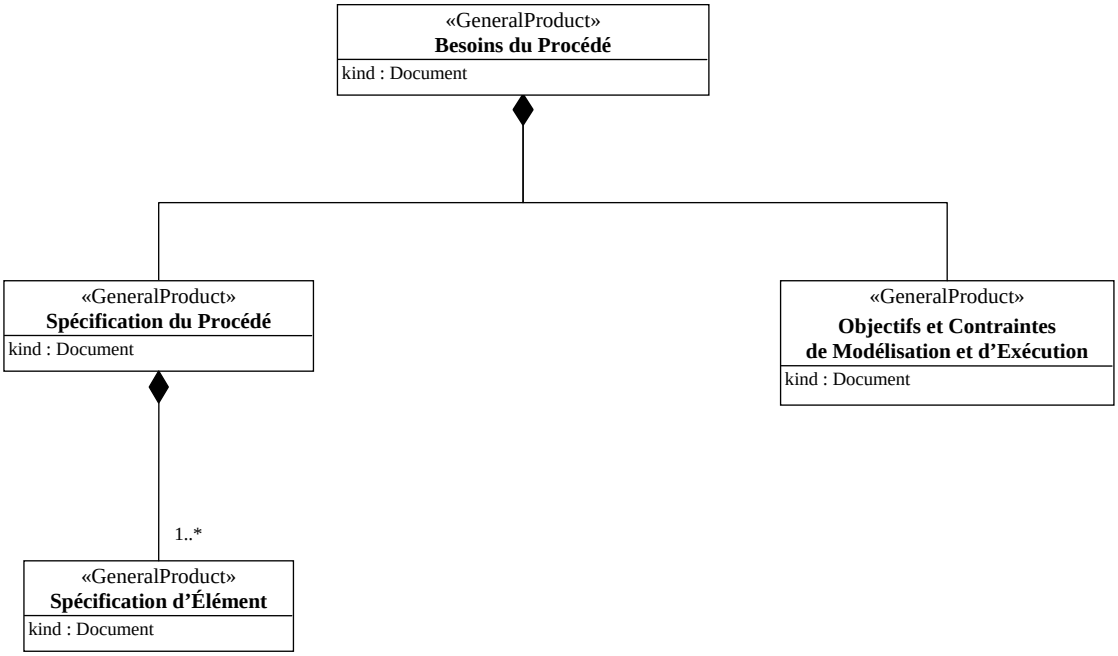


Figure III-19. Structure du produit *Besoins du Procédé*

- **Spécification du Procédé**

Ce document a pour but de spécifier les caractéristiques du procédé à modéliser. Les détails de ce document ne peuvent être précisés que dans le contexte d'un méta-procédé concret. Nous proposons dans le Tableau III-3 les caractéristiques essentielles à déterminer :

Caractéristiques	Signification	Exemple
Niveau de généricité	Le degré de dépendance d'un procédé vis à vis d'une organisation et de ses projets.	▪ *Méthode générale :* démarche indépendante d'organisations ou de projets, qui peut être appliquée à des projets similaires et des organisations ayant des caractéristiques communes. ▪ *Procédé d'organisation* : démarche générale applicable dans le cadre d'une organisation pour un type de projets. ▪ *Procédé spécifique* : démarche concrète qui satisfait les besoins d'un projet concret avec des contraintes de ressources réelles.
Type de procédé	La nature de l'activité globale décrite par le procédé	Développer un logiciel, Maintenir un logiciel, Traiter des informations, etc.
Condition d'application	Les contraintes sur l'environnement où le procédé sera mis en œuvre	Taille et compétence de l'équipe, moyens techniques/matériels de l'organisation, répartition, etc.
Produits fabriqués	La nature des produits fabriqués au cours de l'exécution du procédé	Logiciels d'application, systèmes embarqués, systèmes distribués, etc.
Éléments principaux	L'ensemble des descriptions des éléments primaires[1] du procédé à définir ou à modifier. Dans le cas de modification, cette section indique les éléments concernés.	Procédé « Modéliser un procédé à base de patrons» avec deux tâches principales : *Élaborer les modèles* et *Gérer la base de patrons*

Tableau III-3. Caractéristiques essentielles d'un procédé

- **Objectifs et Contraintes de Modélisation et d'Exécution**

Ce document décrit le but et les contraintes de la modélisation du procédé ainsi que les objectifs attendus de l'exécution du procédé. Ces informations serviront à valider le modèle de procédé élaboré. L'objectif et les contraintes dépendent partiellement des caractéristiques du procédé à définir. Par exemple, si l'on modélise un procédé spécifique, il faut le modéliser au niveau concret et éventuellement avec un LDP formel pour permettre l'automatisation de son exécution.

Le Tableau III-4 montre quelques éléments qui devraient être inclus dans ce document.

[1] En général, à cette étape, seules les tâches principales du procédé seront identifiées, dont la tâche racine du procédé. Des produits et des rôles principaux peuvent être identifiés aussi.

Contrainte	Signification	Exemple
Objectif de Modélisation	Le type de projet de modélisation de procédés	*Créer un nouveau procédé* ou *Modifier un procédé existant*
Niveau de détail du Modèle	Le degré d'abstraction et de granularité du modèle élaboré.	Décrire les étapes du procédé au niveau général, détailler chaque activité du procédé au niveau concret, etc.
Notation de Modélisation	La notation demandée pour représenter le procédé	Les notations formelles (par exemple, SLANG), les notations semi-formelles (par exemple, SPEM), etc.
Métriques de performance du procédé	Les informations recueillies et les critères pour mesurer l'efficacité du procédé	Effort et coût de développement (homme/mois), taille du produit, vitesse de production, etc
Objectifs de définition et de performance	Les aspects importants pour la définition et l'exécution du procédé	▪ Atteindre le niveau 4 du CMMI ▪ Temps/coût de développement, Productivité, etc.

Tableau III-4. Objectifs et Contraintes de modélisation et d'exécution des procédés

Activités

La Figure III-20 montre la structure et le comportement de la tâche *Analyser les Besoins*.

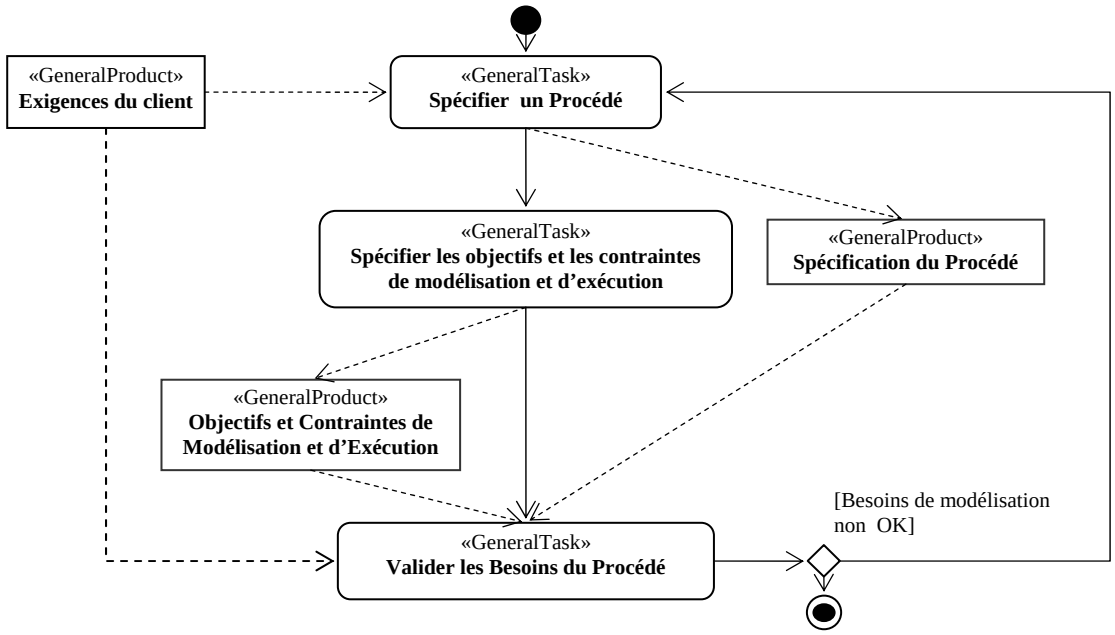


Figure III-20. La tâche *Analyser les Besoins*

L'analyse des besoins commence par la tâche *Spécifier un Procédé* pour spécifier les caractéristiques du procédé à modéliser. Ensuite, la tâche *Spécifier les objectifs et les contraintes de modélisation et d'exécution* est réalisée pour définir les objectifs et les contraintes de la modélisation. Enfin, les produits issus des tâches précédentes sont vérifiés par la tâche *Valider les Besoins du Procédé* pour vérifier si les besoins du procédé sont correctement décrits.

III.3.2. Modéliser un Procédé

Objectif

Cette tâche élabore un modèle de procédé satisfaisant les besoins identifiés dans la tâche *Analyser les Besoins*.

Agents

- Responsable : *Concepteur de Procédé*
- Participant : *Analyste de Procédé*

Produits

- Entrée : *Besoins du Procédé*
- Sortie : *Modèle du Procédé*

Dans cette tâche, le concepteur définit un ensemble de modèles décrivant le procédé satisfaisant les *Besoins du Procédé*. Un *Modèle de Procédé* est utilisé pour décrire un élément de procédé, il comprend deux sous-modèles (Figure III-21):

- **Structure statique d'un élément de procédé**

Ce modèle décrit la structure de l'élément c'est-à-dire sa décomposition (un seul niveau), les relations et les dépendances entre ses sous-éléments (c.f. Chapitre 2, section II.1.6).

Par exemple, la structure statique d'un produit est sa décomposition en sous-produits et les dépendances de type *ProductImpact* entre eux. La structure statique d'une tâche est sa décomposition en sous-tâches ou «steps» et les rôles, les produits relatifs à la tâche.

- **Comportement dynamique d'un élément de procédé**

Ce modèle montre l'aspect dynamique d'un élément de procédé ou des relations de coordination entre éléments de procédé (c.f. Chapitre 2, section II.1.6).

Dans cette thèse, nous n'utilisons la notion de comportement dynamique que pour représenter l'enchaînement des sous-tâches ou « steps» d'une tâche.

On peut considérer un modèle de procédé comme le modèle de la tâche racine du procédé[1]. Cela implique qu'un modèle de procédé contient aussi les modèles des éléments qu'il décrit.

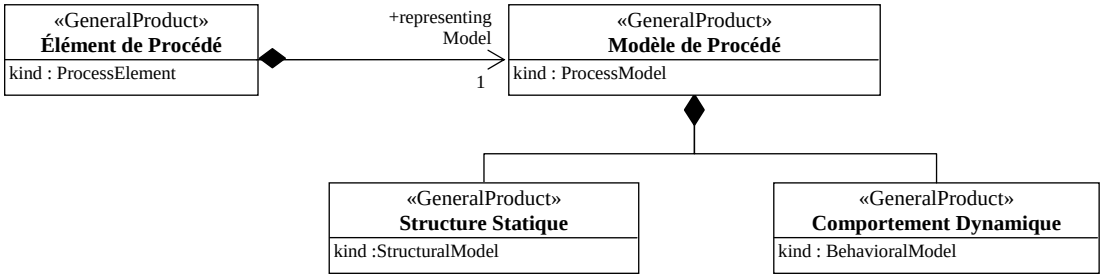


Figure III-21. Structure du produit *Modèle de Procédé*

[1] Dans notre méta-modèle, un procédé est un élément de procédé spécial assimilé à la tâche racine.

Activités

La méta-tâche *Modéliser un Procédé* est composée des quatre sous-tâches suivantes :

- **Définir un Modèle de Procédé** : cette tâche a pour but de définir un nouveau modèle de procédé basé sur la spécification définie par la tâche *Analyser les Besoins*.
- **Modifier un Modèle de Procédé** : cette tâche a pour but de modifier un modèle de procédé existant selon les besoins de changement spécifiés par la tâche *Analyser les Besoins*.
- **Vérifier un Modèle de Procédé** : cette tâche vérifie la correction du modèle défini ou modifié vis-à-vis de la syntaxe et de la sémantique du LDP utilisé.
- **Valider un Modèle de Procédé** : cette tâche vérifie si le modèle défini ou modifié satisfait les objectifs et les contraintes de modélisation définies par la tâche *Analyser les Besoins*.

L'enchaînement de ces tâches est montré sur la Figure III-22. Selon son objectif, la modélisation du procédé commence par la méta-tâche *Définir un Modèle de Procédé* pour définir un nouveau modèle, ou par la méta-tâche *Modifier un Modèle de Procédé* pour modifier un modèle existant. Dans les deux cas, le produit de sortie est un modèle de procédé qui sera en entrée des tâches *Vérifier un Modèle de Procédé* et *Valider un Modèle de Procédé* pour la vérification de sa correction et pour la validation des caractéristiques attendues.

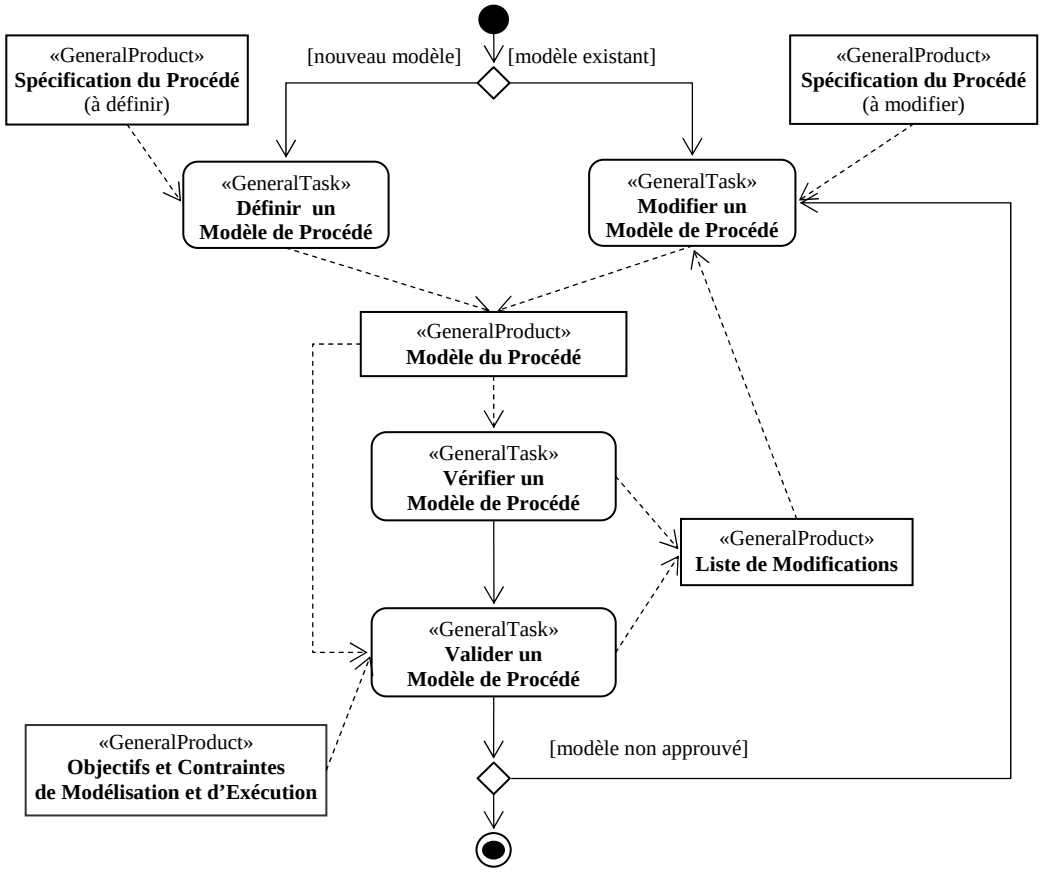


Figure III-22. La tâche *Modéliser le Procédé*

Dans les sous-sections suivantes, nous décrivons les sous-tâches abordées ci-dessus.

III.3.2.1. Définir un Modèle de Procédé

Cette tâche a pour but de définir un nouveau modèle de procédé à partir de la spécification informelle du procédé. Elle est réalisée par le rôle *Concepteur de procédé*.

Nous adoptons l'approche top-down pour définir un procédé. La modélisation du procédé commence toujours par la définition de sa tâche racine (qui représente le procédé lui-même) en utilisant la sous-tâche *Définir un élément de procédé*. Cette sous-tâche a pour but de définir le modèle d'une seule tâche, mais grâce à son exécution récursive (c.f.III.3.2.1.5), le modèle entier du procédé sera généré à partir de sa tâche racine.

Le comportement de la tâche *Définir un Modèle de Procédé* est donc très simple, comme le montre la Figure III-23.

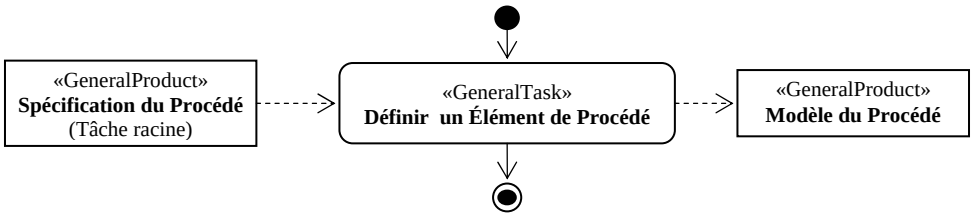


Figure III-23. La tâche *Définir un Modèle de Procédé*

Définir un Élément de procédé

Définir un Élément de Procédé est la tâche centrale du méta-procédé pour créer un élément de procédé et le modèle qui le décrit. C'est une tâche générale qui peut être spécialisée pour définir différents types d'éléments de procédé, soit des tâches, des produits ou des rôles. La Figure III-24 montre les détails de la tâche *Définir un Élément de Procédé*.

D'abord cette tâche crée, via l'action (Step) *Créer un élément,* l'élément de procédé s'il n'existe pas. Notre approche est caractérisée par la réutilisation de patrons de procédé en modélisant des modèles de procédé. Ainsi pour définir un élément, nous essayons d'abord de chercher (dans une *Base de Patrons de Procédé*) un patron qui satisfasse la spécification de l'élément afin de réutiliser sa solution pour générer directement le modèle de l'élément. Cette activité est réalisée par la sous-tâche *Sélectionner un Patron* (c.f. III.3.2.1.1).

S'il existe un patron convenable, la sous-tâche *Appliquer un Patron* élabore le modèle de l'élément en appliquant le modèle de procédé capturé dans la solution du patron choisi (c.f. III.3.2.1.2) Sinon, le modèle de l'élément est construit progressivement. En premier lieu, la tâche *Modéliser l'aspect statique d'un élément* est réalisée pour élaborer la structure statique du modèle de l'élément. Ensuite la tâche *Modéliser l'aspect dynamique d'un élément* est exécutée pour élaborer la partie décrivant son comportement dynamique[1].

Si l'élément en cours de définition est un élément simple, qui n'a pas de composants, sa modélisation s'arrête. Sinon, on passe à la tâche *Définir les Sous-Éléments* (c.f.) pour créer les

[1] A ce jour, PATPRO utilise cette activité pour décrire le comportement de tâche, seul élément où une description de l'aspect dynamique est prévu dans UML-PP. Si le LDP utilisé fournissait un moyen pour décrire les aspects dynamiques des autres types d'éléments de procédé, cette tâche serait applicable à tous les éléments de procédé.

modèles de ses composants. Finalement, le modèle de procédé élaboré est associé à l'élément en cours de définition (Step *Associer un Modèle à un Élément*).

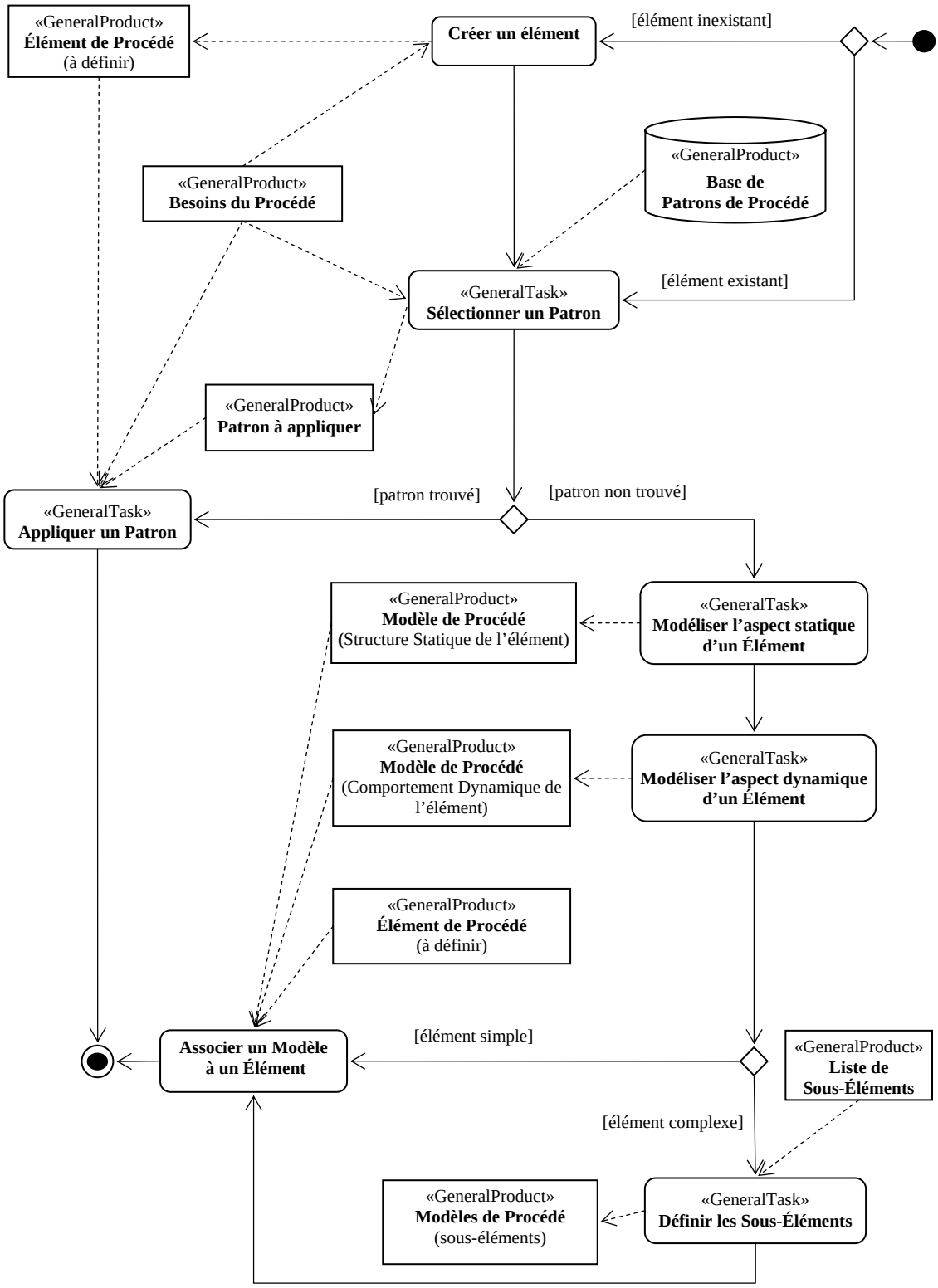


Figure III-24. La tâche *Définir un Élément de procédé*

Les sous-tâches de la tâche *Définir un Élément de procédé* sont présentées ci-dessous dans les sous-sections de III.3.2.1.1 à III.3.2.1.5.

III.3.2.1.1. Sélectionner un Patron

Cette tâche a pour but de sélectionner dans un ensemble de patrons celui dont la spécification est la plus conforme à la spécification de l'élément à définir [1].

A partir de la spécification de l'élément, on cherche des patrons ayant un objectif similaire à celui de l'élément via la tâche *Chercher des Patrons Potentiels*. S'il y a plusieurs patrons satisfaisants, on doit définir le critère (*Définir le critère de conformité*) pour sélectionner le meilleur patron parmi les patrons convenables (*Sélectionner le meilleur Patron*). La Figure III-25 montre le détail de la tâche *Sélectionner un Patron*.

Les deux tâches *Chercher des Patrons Potentiels* et *Sélectionner le meilleur Patron* utilisent respectivement les opérateurs *ProcessPatternSearching()* et *ProcessPatternSelecting()* (c.f.II.1.1) ; elles peuvent être partiellement ou complètement automatisées. La tâche *Définir le critère de conformité* par contre demande l'intervention du concepteur pour préciser les priorités des différents aspects de la spécification de l'élément.

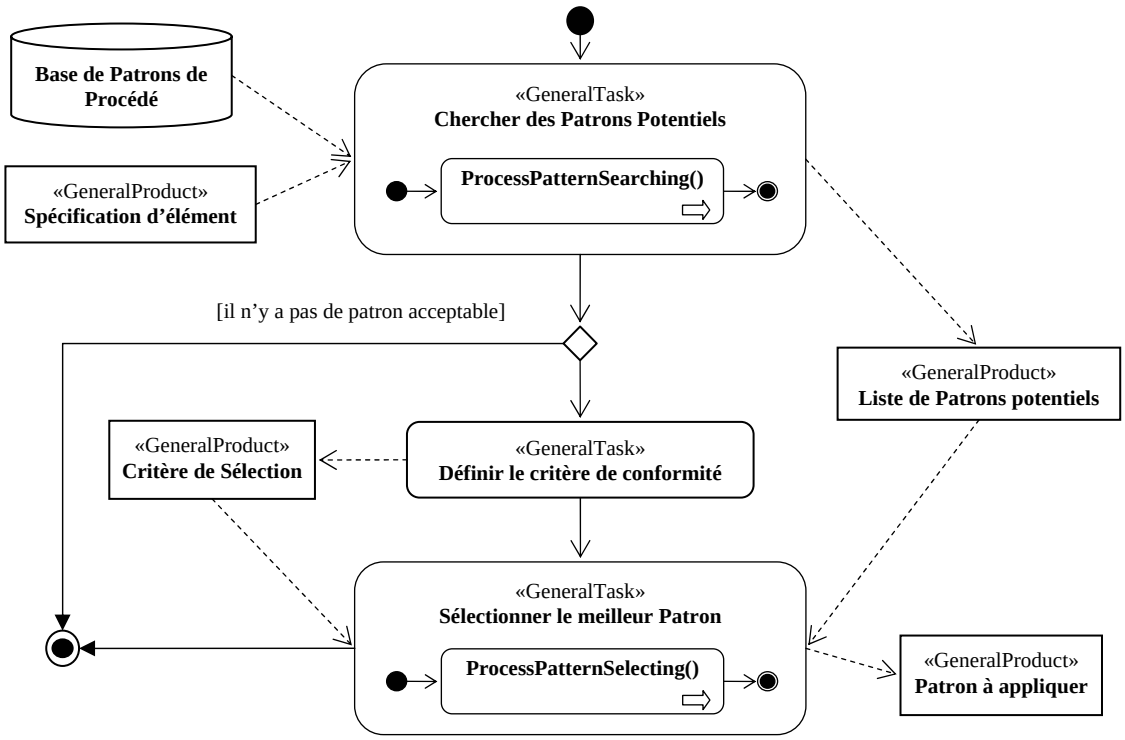


Figure III-25. La tâche *Sélectionner un Patron*

En principe, un patron idéal à choisir devrait correspondre exactement à la spécification de l'élément. S'il n'y a pas de tel patron, on choisira des patrons ayant une spécification similaire à celle de l'élément. Pour pouvoir appliquer ce principe, il faut disposer de critères de comparaison. Dans le contexte de la définition d'un élément de procédé, nous proposons dans le Tableau III-5 un classement des niveaux de conformité d'un patron à la spécification de chaque type d'élément.

[1] La spécification d'un patron comprend son intention et son contexte d'utilisation. La spécification d'un élément de procédé décrit son objectif ou fonction et son contexte. Un patron est considéré comme conforme à la spécification de l'élément si son intention est conforme à l'objectif de l'élément, et si son contexte couvre le contexte de l'élément.

Le degré de conformité d'un patron à un élément de procédé est déterminé par une combinaison de relations entre les aspects du patron (problème, contexte) et les caractéristiques de l'élément.

Dans ce tableau, on dénote :

- e1 = e2 pour exprimer que e1 et e2 sont identiques
- e1 > e2 pour exprimer que e1 contient e2
- e1 < e2 pour exprimer que e1 est inclus dans e2
- e1 ↑ e2 pour exprimer que e1 est une généralisation de e2
- e1 ↓ e2 pour exprimer que e1 est une spécialisation de e2

Pour chaque cellule décrivant une combinaison de relations, il faut l'interpréter de la façon suivante : *[aspect du patron]* doit avoir *[des relations]* avec *[caractéristique de l'élément]*.

Par exemple, en cherchant un patron à appliquer pour définir une tâche, on détermine le degré de conformité du patron à la spécification de la tâche en appliquant les critères proposés dans le Tableau III-5. Pour atteindre le dégré de conformité 2, les condiontions suivantes (déduites de la ligne 2 du Tableau III-5) doivent ête satistaites : le problème du patron est identique à l'objectif de la tâche, le contexte initial du patron est identique à ou inclus dans la précondition de la tâche, le contexte résultant du patron contient la postcondition de la tâche, et le contexte d'application du patron contient les contraintes sur la tâche.

Patron	**Problème**	**Contexte**			**Degré de Conformité**
		Initial	**Résultant**	**d'Application**	
Tâche	**Objectif**	**PréCondition**	**PostCondition**	**Contrainte**	
	=	=	=	=	3
	=	< ou =	= ou >	> ou ↑	2
	↑	< ou = et ↑	= ou > et ↑	> ou ↑	1
Produit	**Type de produit**		**Représentation**		
	=		=		3
	=		↑ou ≠		2
	↑		= ou ↑ ou ≠		1
Rôle	**Responsabilité**		**Compétence**		
	=		=		3
	=		↑ou ≠		2
	↑		= ou ↑ ou ≠		1

Tableau III-5. Critères de comparaison pour déterminer le degré de conformité d'un patron à la spécification d'un élément

Comme nous l'avons discuté dans la section II.1.1, la comparaison de la spécification d'un

patron avec la spécification d'un élément de procédé pour déterminer leur degré de conformité est complexe. Les critères ci-dessus sont donc des propositions qui ont besoin d'être raffinées.

III.3.2.1.2. Appliquer un Patron

Il y a deux façons d'appliquer un patron : l'imiter ou le référencer. L'imitation est utilisée quand le concepteur veut élaborer le modèle détaillé de l'élément sans garder la trace du patron réutilisé, ou quand il a besoin d'adapter la solution du patron sélectionné pour qu'elle soit conforme à la spécification de l'élément. La référence est utilisée si le concepteur veut simplement représenter le lien vers la solution du patron et si la modification de cette solution n'est pas obligatoire.

La Figure III-26 montre le détail de la tâche *Appliquer un Patron*.

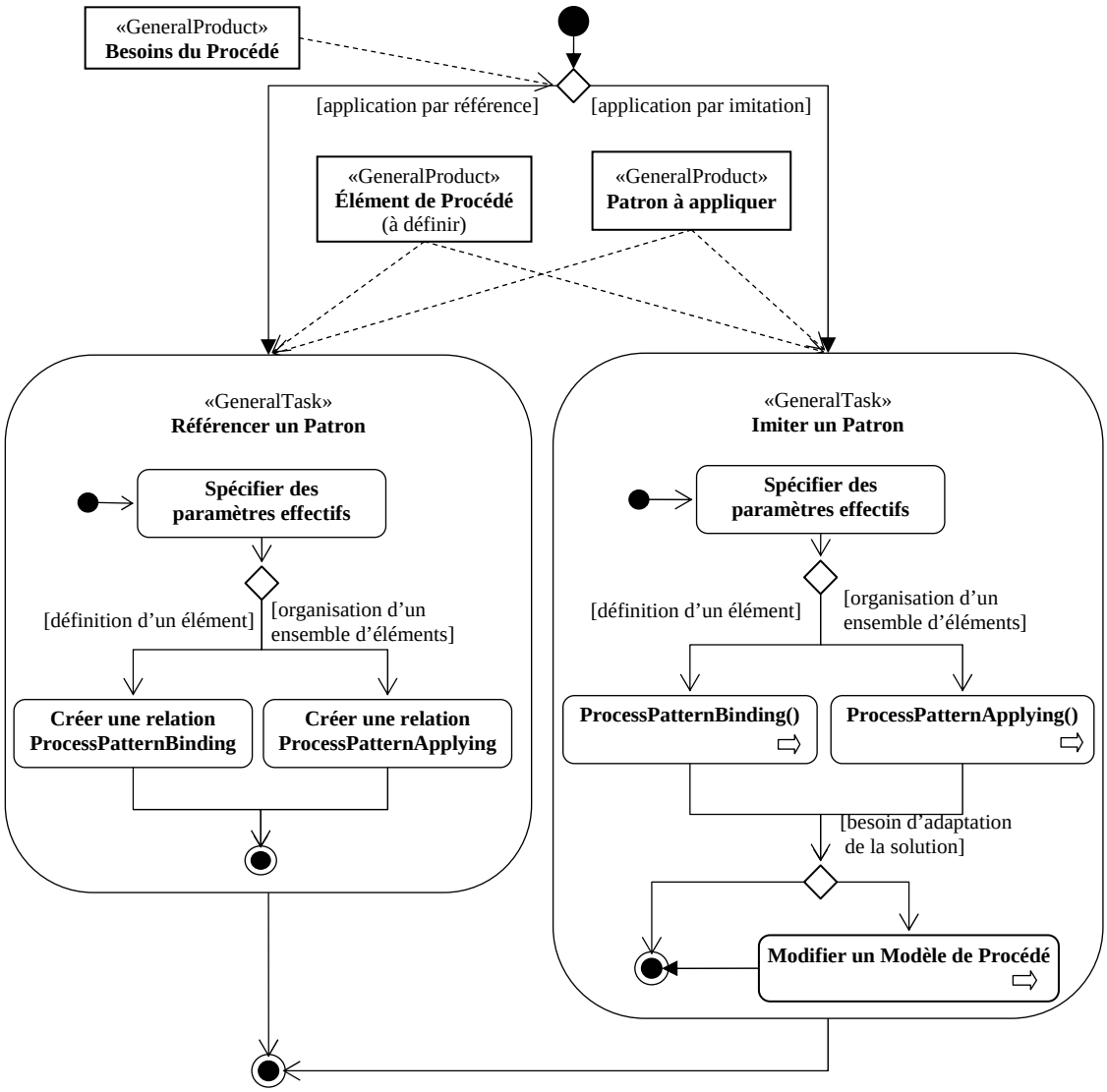


Figure III-26. La tâche *Appliquer un Patron*

La tâche *Référencer un Patron* établit soit une relation *ProcessPatternBinding*, soit une relation *ProcessPatternApplying* entre le patron sélectionné et l'élément (ou l'ensemble d'éléments) modélisé selon l'exigence spécifiée dans le document *Besoins du Procédé*.

La tâche *Imiter un Patron*, quant à elle, exécute selon le besoin, soit l'opérateur *ProcessPatternBinding*, soit l'opérateur *ProcessPatternApplying* pour dupliquer la solution du patron et le réutiliser en tant que modèle décrivant l'élément. Ce modèle pourra être modifié ensuite par la tâche *Modifier un Modèle de Procédé* (c.f. III.3.2.2) pour l'adapter au besoin spécifique de l'élément.

III.3.2.1.3. Modéliser l'aspect statique d'un Élément

Cette tâche a pour but d'élaborer la partie *Structure Statique* d'un modèle de procédé. A partir de la spécification de l'élément, le concepteur identifie ses composants et les relations entre eux (*Décomposer un Élément*). Le détail de cette décomposition dépend du type d'élément à définir[1]. Ensuite, le concepteur utilise ces informations pour créer un modèle structurel (*Créer un Diagramme de Structure Statique*).

La Figure III-27 montre la tâche *Modéliser l'aspect statique d'un Élément*.

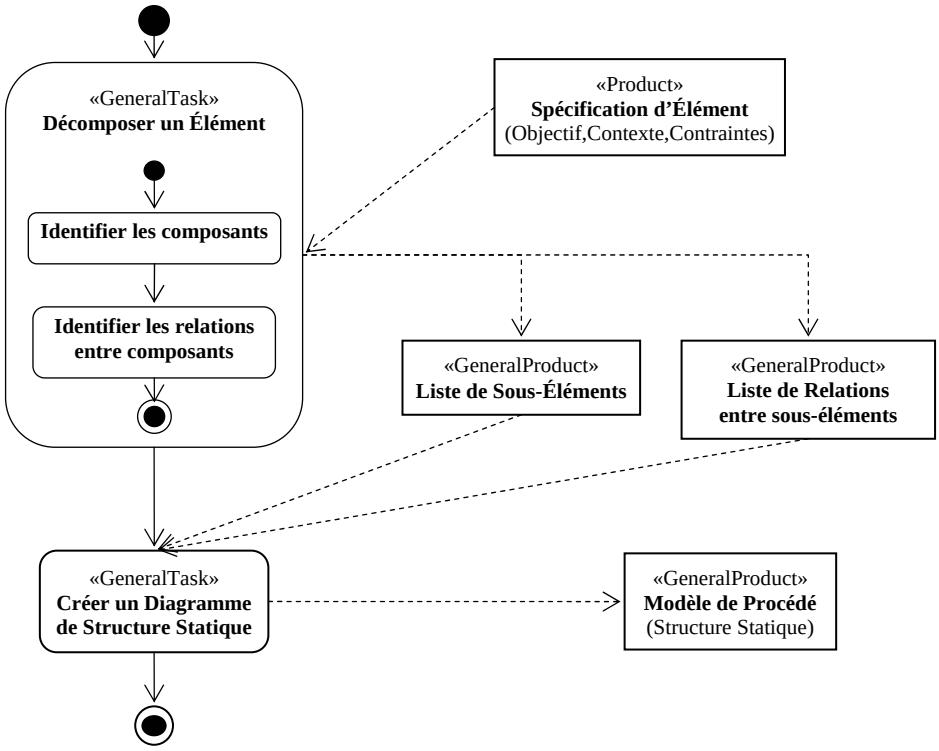


Figure III-27. La tâche *Modéliser l'aspect statique d'un Élément*

III.3.2.1.4. Modéliser l'aspect dynamique d'un Élément

En fait cette tâche a pour but d'élaborer le modèle comportemental d'une tâche. D'abord on identifie les actions qui composent la tâche (*Identifier les Actions d'une Tâche*). Une action peut être une activité atomique (*Step*) ou l'appel à une autre activité complexe. Ensuite, des relations exprimant l'ordre d'exécution de ces actions sont identifiées (*Identifier les Relations entre*

[1] Comme nous adoptons l'approche top-down pour élaborer un modèle de procédé, à cette étape, les relations de l'élément avec les éléments du même niveau de décomposition ont été décrites dans le modèle de leur élément supérieur.

actions). Finalement, un diagramme d'activité représentant la partie dynamique du modèle est créé (*Créer un Diagramme de Comportement Dynamique*).

La Figure III-28 montre le détail de la méta-tâche *Modéliser l'aspect dynamique d'un élément*.

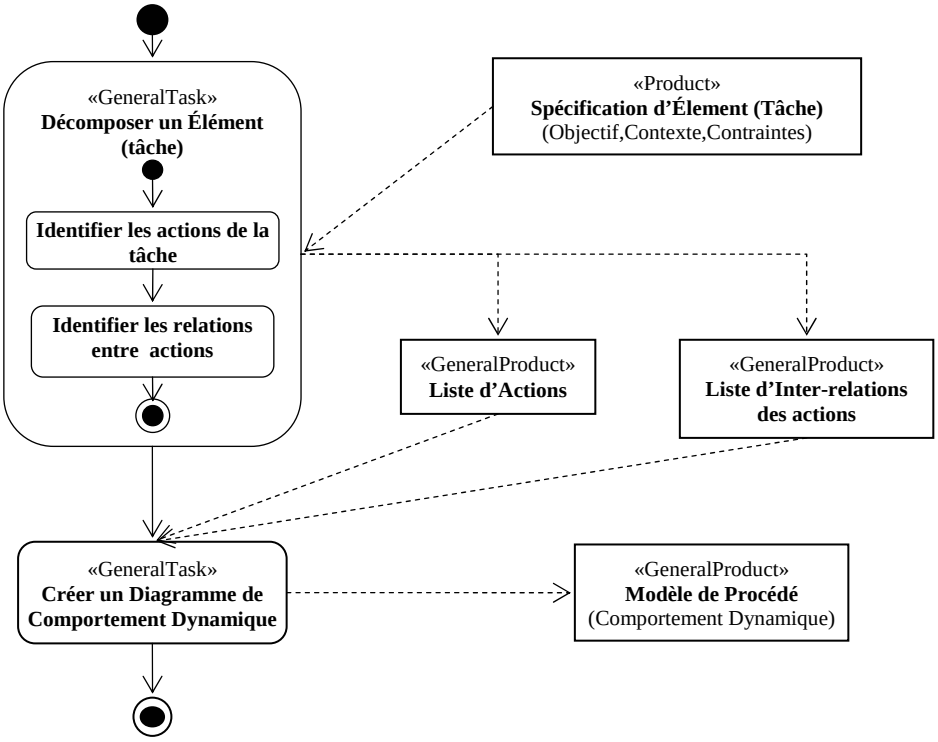


Figure III-28. La tâche *Modéliser l'aspect dynamique d'un Élément*

III.3.2.1.5. Définir les Sous-Éléments

Cette tâche a pour but d'élaborer les modèles décrivant les composants d'un élément de procédé complexe. La tâche *Définir un Élément* est utilisée récursivement pour créer le modèle de chaque composant dans la liste des sous-éléments d'un élément complexe (Figure III-29).

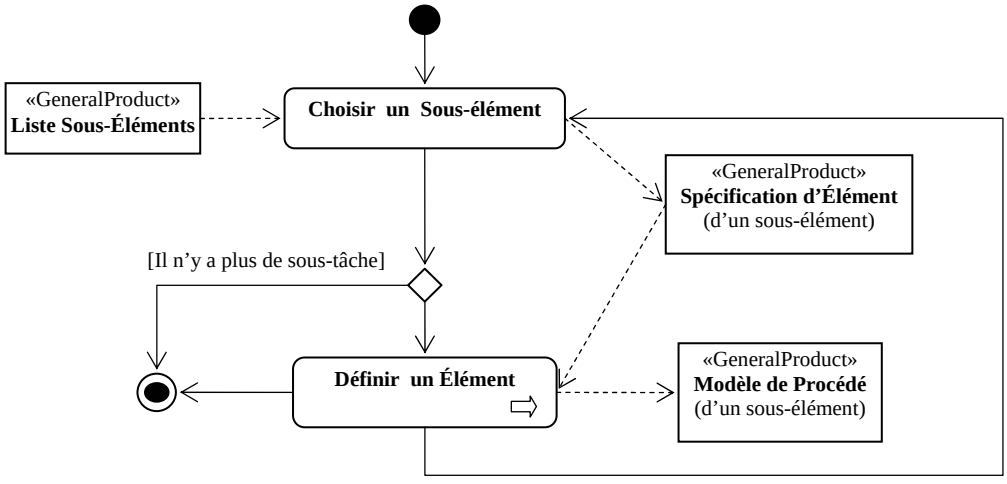


Figure III-29. La tâche *Définir les Sous-Éléments*

III.3.2.2. Modifier un Modèle de Procédé

Cette (méta-)tâche réalisée par les concepteurs de procédé a pour but de modifier un modèle de procédé existant selon des besoins spécifiés dans le document *Besoins du Procédé*.

Comme nous l'avons expliqué dans le contexte de la tâche *Analyser les besoins* (c.f. III.3.1), si l'objectif de la modélisation est la modification du procédé, le document *Besoins du Procédé* contient une liste de changements à faire, chaque changement pouvant concerner un ou plusieurs éléments du modèle de procédé. La modification d'un modèle de procédé (Figure III-30) est donc le traitement de cette liste de besoins de modification, traitement qui consiste à exécuter autant de fois que nécessaire la sous-tâche *Traiter un Besoin de modification*.

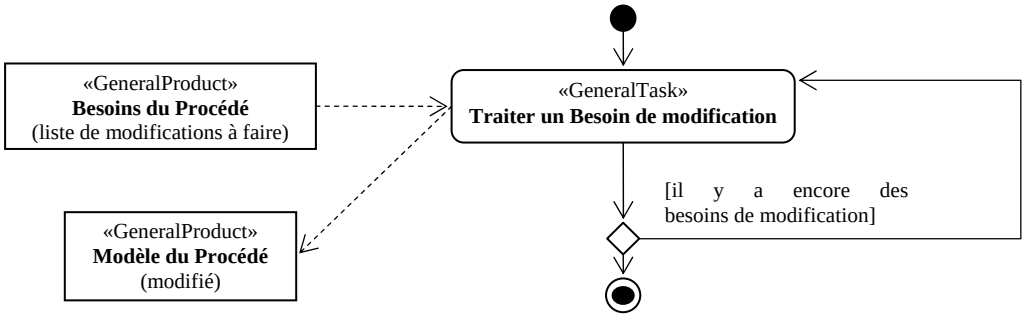


Figure III-30. La tâche *Modifier un Modèle de Procédé*

Traiter un Besoin de modification

La modification d'un modèle de procédé peut être effectuée pour une des raisons suivantes :

- *Redéfinition d'élément* : c'est le cas où le concepteur veut redéfinir un élément, il a donc besoin de changer le modèle décrivant l'élément.
- *Adaptation de Modèle* : c'est le besoin d'adapter un modèle général à un contexte plus spécifique pour se conformer aux caractéristiques d'une organisation ou d'un projet concret.
- *Amélioration de Modèle :* c'est le besoin de modification d'un modèle pour obtenir un modèle plus raffiné ou un modèle améliorant les performance du procédé. Il s'agit dans ce cas soit de la correction d'erreurs de modélisation, soit de la réorganisation du modèle pour améliorer certaines faiblesses de structure du procédé.

Les opérations de modification peuvent varier pour chaque besoin de modélisation. Le Tableau III-6 montre une synthèse des types de modification à faire sur un modèle de procédé. Dans ce tableau, nous réutilisons la notation proposée dans la section II pour exprimer la signification des types de modification. Rappelons que:

- $e \in E$ est un élément de procédé
- $pm \in PM$ est un modèle de procédé, $pm = (E_{pm}, R_{pm})$, où E_{pm} et R_{pm} sont des ensembles d'éléments et de relations de *pm*.
- *e* est décrit par le modèle *pm*, dénoté par $e \leftarrow pm$

N°	Type de modification	Opération	Éléments concernés	Motivation
1	Redéfinir un élément e	Changer pm par un autre modèle	L'élément e Le modèle pm entier	Redéfinition
2	Spécialiser un modèle pm	Rendre pm plus spécifique en substituant les éléments de pm par d'autres plus spécifiques	Les éléments $e_i \in E_{pm}$	Adaptation Amélioration
3	Raffiner un modèle pm	Modifier les éléments e_i pour les rendre conformes à un besoin donné.	Les éléments $e_i \in E_{pm}$	Adaptation Amélioration
4	Simplifier un modèle pm	Supprimer des éléments e_i ou des relations r_i de pm	Les éléments $e_i \in E_{pm}$ Les relations $r_i \in E_{pm}$	Adaptation Amélioration
5	Enrichir un modèle pm	Ajouter de nouveaux éléments ne_i ou de nouvelles relations nr_i au pm	Les éléments $e_i \in E_{pm}$ Les relations $r_i \in E_{pm}$	Adaptation Amélioration
6	Restructurer un modèle pm	Modifier des relations entre des éléments e_i de pm	Les relations $r_i \in E_{pm}$	Amélioration

Tableau III-6. Types de modification de modèle de procédé

Il est impossible de définir un comportement détaillé de la tâche *Traiter un Besoin de modification* car ce comportement dépend des besoins de modification concrets. Ainsi, dans la Figure III-31, nous décrivons un ordre d'exécution général de cette tâche selon les besoins[1] définis dans le Tableau III-6.

Comme dans la définition d'un modèle, le concepteur en charge de la modification d'un modèle peut aussi réutiliser des patrons de procédé pour réduire le temps de modélisation et profiter des connaissances de procédé éprouvées. Cependant, en général, la réutilisation de patrons est seulement applicable aux types de modification (1), (2) ou (6). Les autres types de modification nécessitent de manipuler le modèle.

La tâche *Traiter un Besoin de modification* commence donc par une analyse du type de modification demandé. Si c'est un des types (1), (2) ou (6), la tâche *Sélectionner un Patron* (c.f.III.3.2.1.1) est invoquée pour chercher un patron convenable. Si un patron a été trouvé, il est appliqué par l'intermédiaire de la tâche *Appliquer un Patron* (c.f.III.3.2.1.2).

Si aucun patron approprié n'est trouvé, ou si les types de modification sont (3), (4), ou (5), la tâche *Manipuler un Modèle* est activée. Cette tâche identifie d'abord le type de modification à faire (Step *Identifier un Besoin de Modification*), puis elle applique les opérateurs d'adaptation (*ElementAdding*, *ElementDeleting*, etc.) pour réaliser la modification.

La sélection des patrons à réutiliser pour modifier un modèle n'est pas évidente car en général, les besoins de modification et les problèmes des patrons adressant de tels besoins sont difficiles à formuler. Par conséquent, une automatisation totale pour la sélection des patrons est difficilement envisageable.

[1] Un besoin de modification de modèle peut être une combinaison de plusieurs des types de modification présentés ci-dessus.

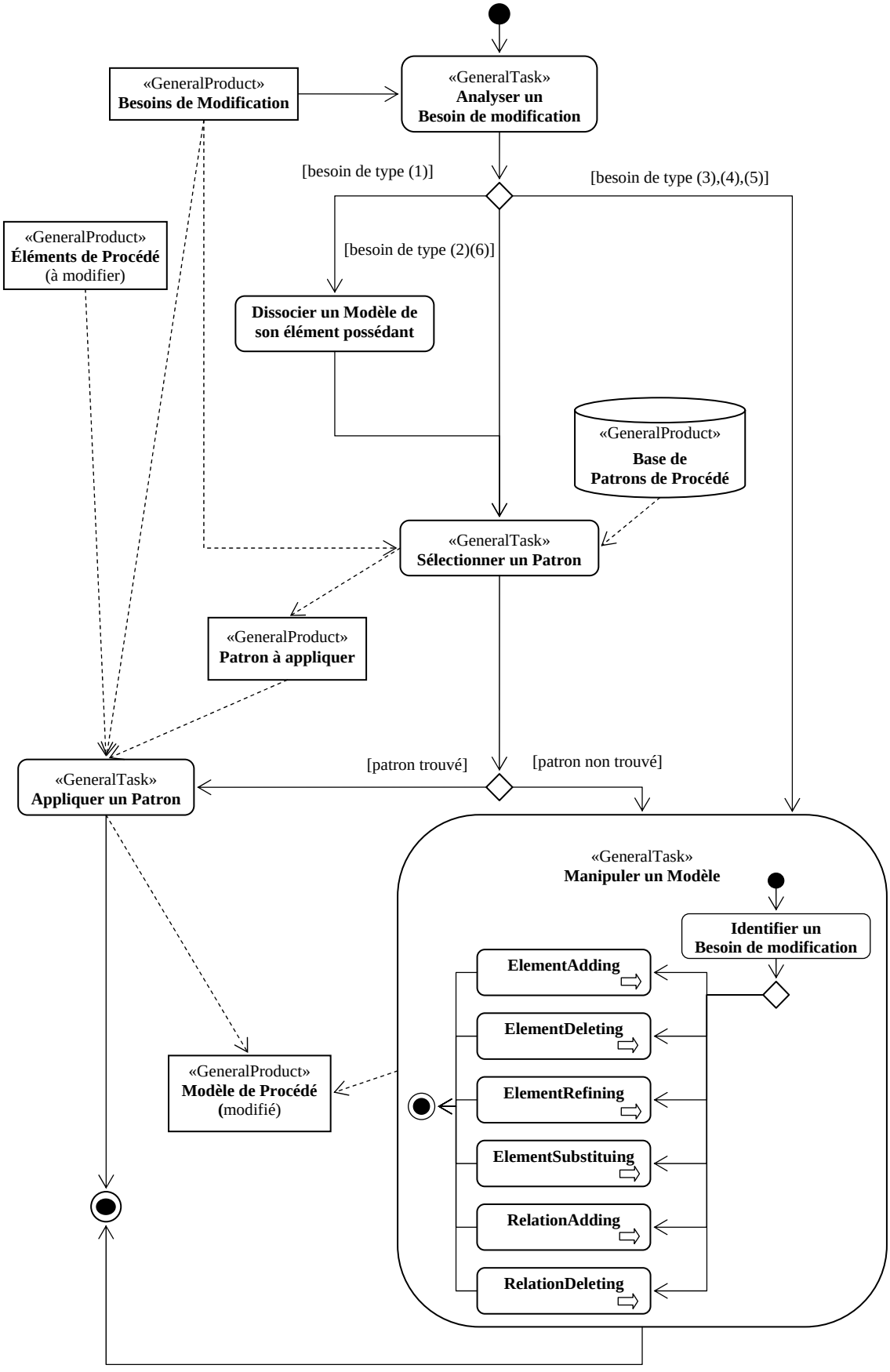


Figure III-31. La tâche *Traiter un Besoin de modification*

III.3.2.3. Vérifier un Modèle de Procédé

La méta-tâche *Vérifier un Modèle de Procédé* a pour but d'assurer qu'un modèle de procédé est cohérent, adéquat et conforme à la syntaxe et à la sémantique du LDP utilisé pour le représenter. C'est le rôle *Concepteur de Procédé* qui est responsable de cette tâche.

Dans notre méta-modèle, le modèle d'un procédé est celui qui est associé à la tâche racine du procédé — c'est-à-dire l'élément représentant le procédé lui-même. Ce modèle se compose d'éléments, chacun de ces éléments étant à son tour associé à un modèle. Par conséquent, la vérification d'un modèle est une activité récursive.

La Figure III-32 montre les actions principales de la tâche *Vérifier un Modèle de Procédé*.

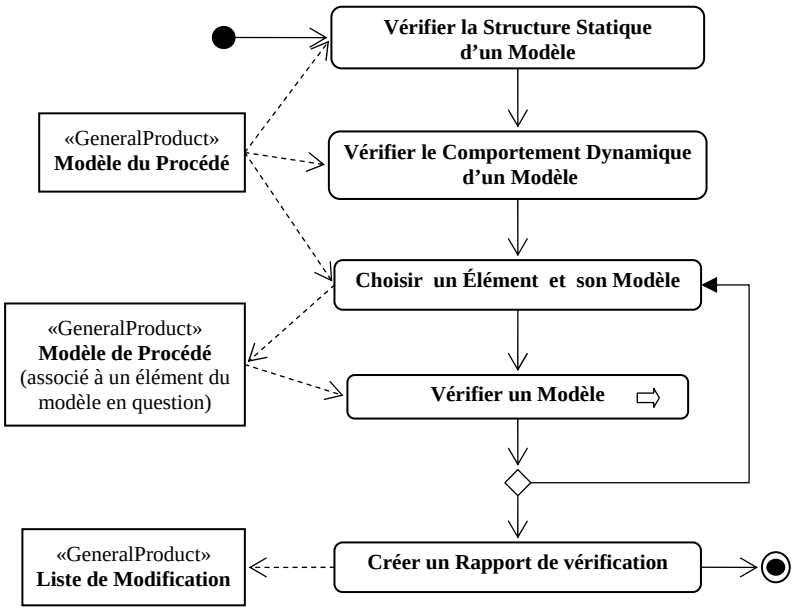


Figure III-32. La tâche *Vérifier un Modèle de Procédé*

III.3.2.4. Valider un Modèle de Procédé

Cette méta-tâche est réalisée par le rôle *Analyste de Procédé*. Son objectif est de vérifier si les objectifs et les contraintes de modélisation imposés sur le modèle sont satisfaits.

La Figure III-33 montre les actions associées à cette tâche.

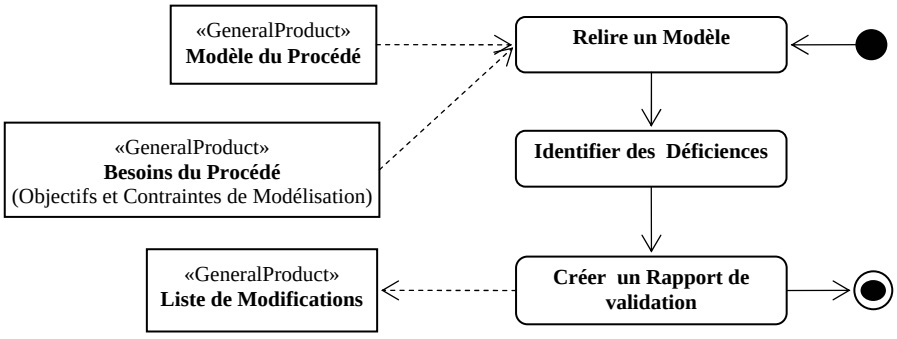


Figure III-33. La tâche *Valider un Modèle de Procédé*

III.3.3. Simuler un Procédé

Objectif

Cette tâche instancie le modèle défini et exécute l'instance obtenue en enregistrant des informations quantitatives et qualitatives sur la performance du procédé.

Dans notre méta-procédé, la simulation est considérée comme un moyen pour tester un procédé. En utilisant des outils appropriés durant la simulation, on peut identifier certains problèmes (comme l'interblocage, la famine, etc.) et observer le comportement dynamique (compteurs de fréquence, flot d'objets, retards...) du procédé.

Agents

- Responsable : *Opérateur de Procédé*

Produits

- Entrée : *Modèle du Procédé*
- Sortie : *Rapport de Performance du Procédé*

Activités

La simulation d'un procédé concerne la création d'une instance du modèle (*Instancier un Modèle*) et l'exécution de l'instance obtenue (*Exécuter une instance de Procédé*) en lui assignant des ressources réelles (outils, personnels, etc.). Durant l'exécution, des informations concernant la performance du procédé sont enregistrées et passées ensuite à l'activité *Créer un Rapport de Performance de procédé* pour créer le rapport de la simulation.

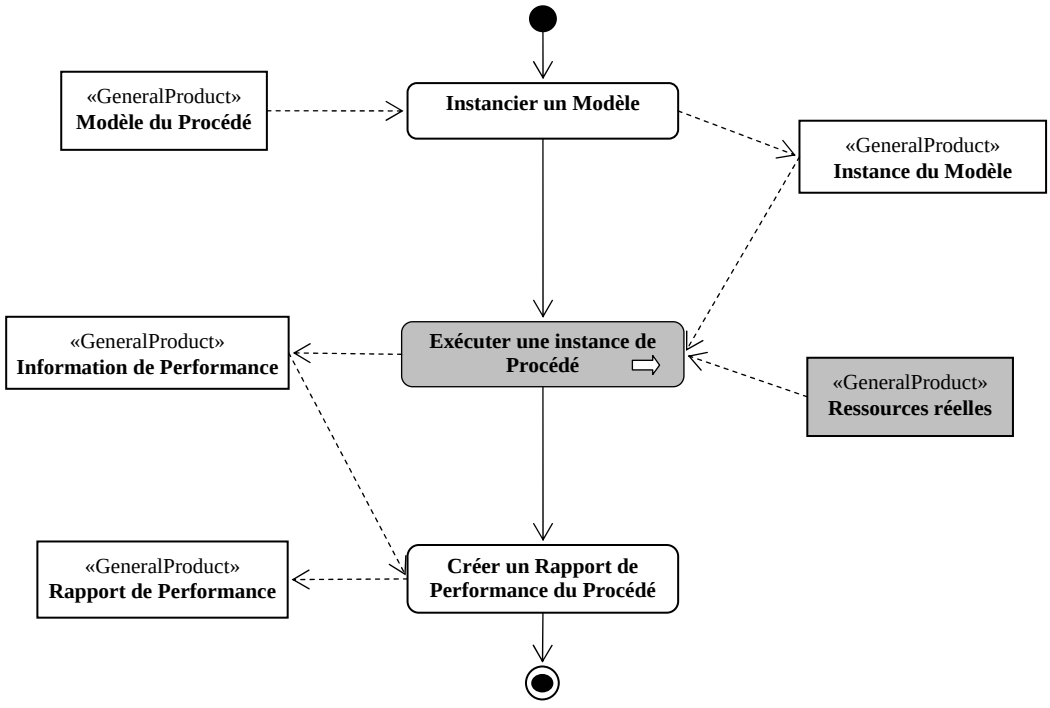


Figure III-34. La tâche *Simuler un Procédé*

III.3.4. Évaluer un Procédé

Objectif

Cette tâche analyse le rapport de performance d'un procédé pour savoir si l'exécution a respecté les contraintes imposées sur sa performance et identifier les déficiences éventuelles. Le résultat d'évaluation sert à proposer de nouvelles exigences pour l'amélioration du procédé.

Agents

- Responsable : *Analyste de Procédé*

Produits

- Entrée : *Rapport de Performance du Procédé*
  Besoins du Procédé (Objectifs et Contraintes de Performance)
- Sortie : *Rapport d'Évaluation du Procédé*

Activités

La Figure III-35 montre le comportement de la tâche *Évaluer un Procédé*. L'activité principale de cette évaluation est la sous-tâche *Analyser un Rapport de Performance* qui compare la performance du procédé avec les objectifs et les contraintes définies dans *Besoins du Procédé*. Il y a plusieurs techniques pour implémenter cette tâche, mais nous ne les détaillons pas ici car elles dépendent du contexte d'un projet de modélisation concret.

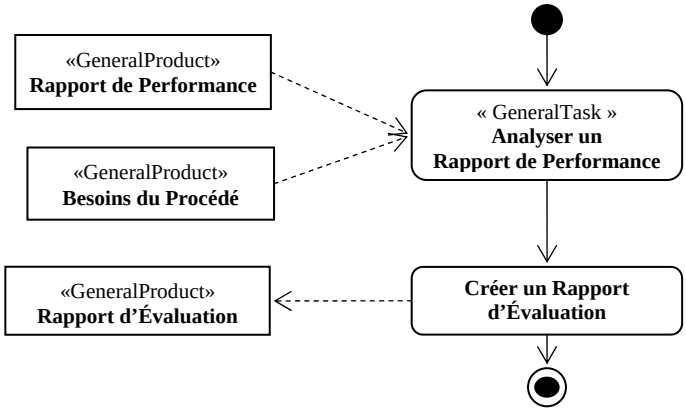


Figure III-35. La tâche *Évaluer un Procédé*

IV. CONCLUSION

Nous avons proposé dans ce chapitre une méthode de modélisation de procédés à base de patrons réutilisables. Notre méthode comporte deux constituants principaux : un ensemble d'opérateurs de réutilisation de patrons de procédé et un méta-procédé guidant la modélisation de procédés en réutilisant des patrons de procédé.

Nous avons défini la sémantique opérationnelle des opérateurs de réutilisation pour permettre leur (partiel) automatisation. Pour exprimer formellement cette sémantique, nous avons décris ces opérateurs en langage Kermeta sous forme de méta-opérations décrivant les

comportements des concepts concernant les patrons de procédé (voir Annexe A). Cela nous permet d'ajouter au méta-modèle UML-PP l'exécutabilité des concepts relatifs à la réutilisation de patrons de procédé.

La définition de cette sémantique opérationnelle clarifie les opérations à faire pour réutiliser les patrons. Dans cette définition, nous avons également identifié et discuté différents scénarios à considérer pour traiter des conflits et des incohérences qui peuvent se produire lors de l'application d'un patron à un modèle de procédé existant. Cependant, notre proposition a besoin d'être approfondie car les problèmes posés soulèvent la difficile question de la fusion de modèles, qui est un challenge du domaine de l'IDM.

Sur le plan méthodologique, nous avons développé le méta-procédé PATPRO comportant des tâches de modélisation de procédés qui favorisent la réutilisation de patrons. PATPRO est un méta-procédé à grains fins qui fournit un guidage systématique aux concepteurs de procédé. Pourtant, pour être plus flexible et efficace, ce méta-procédé devra dans l'avenir prendre en compte l'adaptation dynamique des modèles de procédé.

La méthode que nous avons proposée dans ce chapitre est implémentée sous la forme d'un prototype permettant de modéliser des procédés à base de patrons réutilisables. Ce prototype est présenté dans le Chapitre IV.

CHAPITRE IV.

RÉALISATION DE L'ENVIRONNEMENT PATPRO-MOD

Dans le chapitre II de cette thèse, nous avons présenté une formalisation du concept de patron de procédé intégrée dans le méta-modèle UML-PP. Dans le chapitre III, nous avons décrit une méthode de modélisation de procédés à base de patrons réutilisables, avec notamment le méta-procédé PAT-PRO. L'objectif de ce chapitre est maintenant de décrire l'implémentation d'un environnement de modélisation de procédés, nommé PATPRO-MOD, qui supporte la méthode proposée.

Nous décrivons dans la section 1 les fonctionnalités et l'architecture de l'environnement développé, ainsi que la plateforme de développement utilisée. Dans la section 2 nous introduisons l'étude de cas choisie, et dans les sections 3 et 4, nous illustrons l'utilisation de l'outil PATPRO-MOD sur cette étude de cas. Plus précisément, la section 3 décrit le module de PATPRO-MOD dédié à la gestion des patrons de procédé, et la section 4 présente le module qui supporte la modélisation de procédés. Nous concluons le chapitre par une description de l'état d'avancement de la réalisation.

I. PRINCIPE DE L'IMPLÉMENTATION

I.1. ARCHITECTURE DE L'ENVIRONNEMENT PATPRO-MOD

Pour valider les propositions faites dans cette thèse, nous avons réalisé un outil de modélisation de procédés qui permet :

- d'élaborer des modèles de procédé décrits en UML-PP
- de réutiliser des patrons de procédé au cours de la modélisation de procédés

Pour cela, nous avons développé le prototype PATPRO-MOD composé des deux modules principaux suivants :

- Un module de *gestion de base de patrons* de procédé qui permet de créer, stocker, naviguer et manipuler des patrons de procédé.
- Un module de *modélisation de procédés* qui permet de créer, manipuler et sauvegarder les modèles de procédé.

Ces deux modules ne sont bien sûr pas totalement séparés. En effet, pour créer un patron de procédé il faut travailler sur le modèle capturé dans sa solution (en utilisant *l'éditeur graphique des diagrammes de procédé* du module de modélisation de procédés). De même, pour créer un modèle de procédé on peut réutiliser des patrons de procédé (en utilisant des *opérateurs de recherche et d'imitation de patron*).

La Figure IV-1 montre l'architecture globale du prototype PATPRO-MOD.

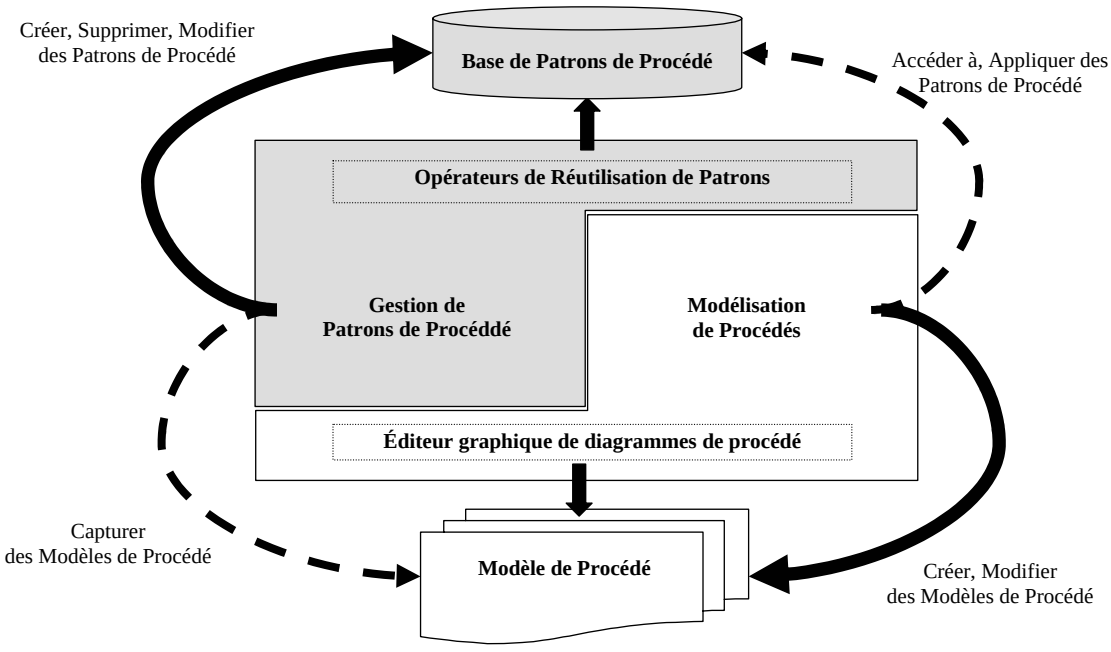


Figure IV-1. Architecture de l'environnement PATPRO-MOD

I.1.1. Gestion des Patrons de Procédé

Le but de ce module est de gérer la base de patrons de procédé du système[1]. L'utilisateur principal de ce module est le rôle *PatternDesigner* qui est responsable de la base de patrons. En pratique, ce module gère une liste de patrons et permet aux concepteurs de réaliser les opérations suivantes : créer un nouveau patron, sélectionner un patron, modifier un patron et supprimer un patron (Figure IV-2).

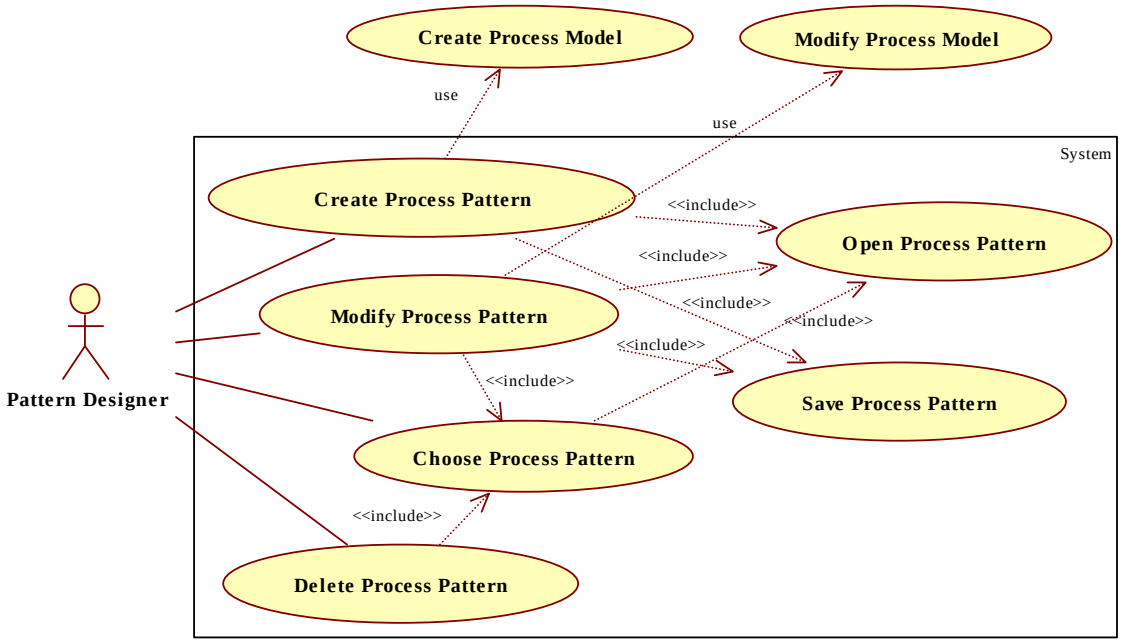


Figure IV-2. Fonctionnalités du module *Gestion des Patrons de Procédé*

Dans la section III, nous présentons l'interface et l'utilisation de ce module via une étude de cas

I.1.2. Modélisation de procédés

Ce module permet d'élaborer des modèles de procédé en réutilisant éventuellement des patrons de procédé. Son utilisateur principal est donc le concepteur de procédé (*ProcessDesigner*). Ce module offre les fonctions suivantes : création d'un modèle de procédé, modification d'un modèle de procédé et dépliage d'un modèle de procédé (Figure IV-3).

Les fonctions de création et de modification ont évidemment pour but de créer et de modifier un modèle de procédé. La fonction de dépliage permet de générer automatiquement un modèle cible à partir d'un modèle source contenant des relations d'application de patrons (c'est-à-dire les relations *ProcessPatternBinding* et *ProcessPatternApplying*).

L'interface et l'utilisation de ce module sont également présentées dans la section III.

[1] Pour simplifier dans un premier temps le prototype, nous supposons que l'environnement dispose d'une seule base de patrons. Par conséquent, la fonction d'administration de bases de patrons peut être ignorée.

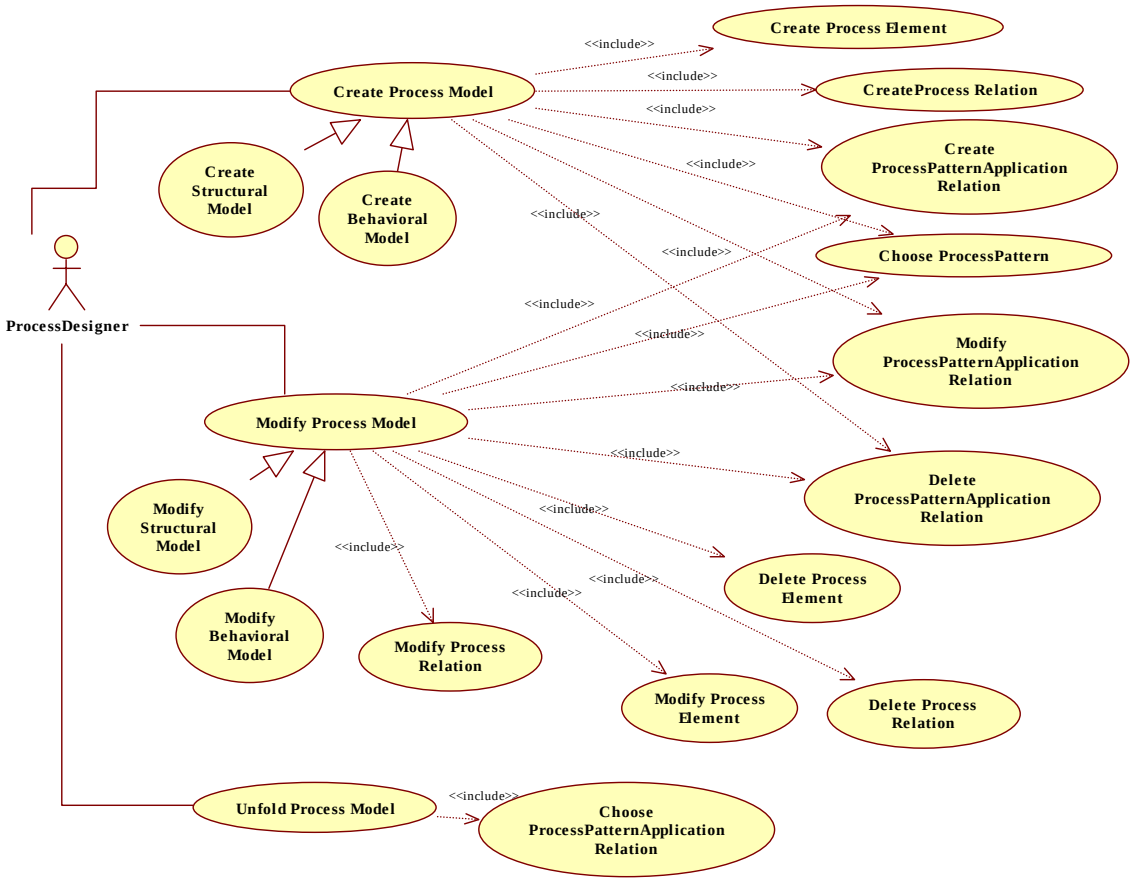


Figure IV-3. Fonctionnalités du module *Modélisation de procédés*

L'environnement PATPRO-MOD s'appuie sur le méta-procédé PATPRO pour guider les concepteurs de procédé dans leur travail. Autrement dit, le méta-procédé est codé dans l'environnement afin de proposer au concepteur un ordre d'exécution des activités de modélisation.

I.2. ENVIRONNEMENT DE DÉVELOPPEMENT

Dans un premier temps, nous avons développé ce prototype en *C#* sous l'environnement *Sharpdevelop2.2* pour la plateforme Windows [CSharp06]. Le choix d'un langage de programmation objet était naturel car il facilite l'implémentation du méta-modèle UML-PP, qui a été élaboré lui aussi selon l'approche orientée-objet. Le langage *C#* a été choisi pour satisfaire des contraintes de développement liées au fait que ce travail de réalisation a été mené en collaboration avec notre équipe de recherche à l'Université des Sciences Naturelles de HochiMinh Ville au Vietnam [Pham08].

II. ÉTUDE DE CAS

Pour illustrer la réutilisation de patrons durant la modélisation de procédés en utilisant l'outil PATPRO-MOD, nous nous appuyons sur la modélisation du procédé VUP (*View based*

Unified Process), un procédé permettant la construction de composants de conception multivue VUML. VUML (*View based Unified Modeling Language*) [Nassar05] est un profil UML qui offre la notion de classe multivue pour permettre la modélisation d'un système selon différents points de vues.

Le but de l'étude de cas est de montrer comment la modélisation d'un procédé à base de patrons réutilisables peut être réalisée sous l'environnement PATPRO-MOD. Nous ne détaillons pas ici toutes les étapes de la modélisation avec VUP, mais seulement certaines activités permettant d'illustrer les fonctionnalités du prototype PATPRO-MOD.

Dans la suite, nous présentons d'abord la base de patrons de procédé utilisée dans cette étude de cas (section II.1), puis nous introduisons le procédé VUP (section II.2).

II.1. BASE DE PATRONS DE PROCÉDÉ

Le Tableau IV-1 montre la base de patrons de procédé que nous utilisons dans cette étude de cas. Pour chaque patron, nous décrivons son nom, le problème associé, le niveau d'abstraction, les contextes initial et résultant.

Patron	Problème	Niveau d'abstraction	Contexte initial	Contexte résultant
Requirements Elicitation	Analyze and Describe System's Requirements	General	Business System Description available	System Requirement created
RUP Requirements	Analyze and Describe System's Requirements in UML	Concrete	Business System Description available	Vision approved UseCase Model approved Glossary approved
RUP Requirement Document	Describe the structure of a RUP requirement document	Concrete		RUP Requirement Document
RUP Analyze the Problem	Identify Requirements to be solved	Concrete	BusinessUC Model Bussiness Glossary New system	Actors outlined Glossary captured Vision created
RUP Capture a Common Vocabulary	Create Glossary in RUP format	Concrete		Glossary captured presented in RUP format
RUP Find Actors and Use Cases	Create UC Model	Concrete	Glossary captured Vision created	UC Model created
RUP Develop Vision	Define goals of system	Concrete	UC Model created	Vision created
RUP Understand Stakeholder Needs	Identify Requirements to be solved	Concrete	BusinessUC Model Bussiness Glossary Existing system	UC Model created Glossary captured Vision created
RUP Define the System	Define the system to be developed	Concrete	Glossary captured Vision created	UCModel refined Glossary refined Vision refined
RUP Use-Case Analysis	Elaborate Analysis Model	Concrete	UC Model created	Analysis Class created Analysis Model created Use-Case Realization created
Working with List	Handle each element of a list with the same activity	Absract	List available	Elements of List handled

Tableau IV-1. Base de patrons de procédé pour l'étude de cas

Dans un premier temps, nous n'utilisons qu'une liste simple pour organiser la base des patrons. Dans cette liste on peut trouver des patrons aux différents niveaux d'abstraction et pour différents problèmes. Nous avons ainsi le patron abstrait *Working with List* décrivant une itération d'une activité donnée sur des éléments d'une liste d'objets quelconque. Nous avons également des patrons généraux, comme le patron *Requirements Elicitation* décrivant les activités générales à réaliser pour analyser les exigences d'un système. Nous avons aussi des patrons concrets, tels que le patron *RUP Requirements* décrivant les activités de l'approche RUP [Kruchten03] à réaliser pour analyser les exigences d'un système et les décrire sous forme d'un modèle de cas d'utilisation en UML. Les patrons concrets sont extraits du procédé RUP, et plus particulièrement de la discipline *Requirement* et *Analysis and Design* du RUP, car la plupart d'entre eux seront réutilisés par le procédé VUP.

Dans la section III nous illustrons la création de cette base des patrons par l'outil PATPRO-MOD.

II.2. SPÉCIFICATION DU PROCÉDÉ VUP

Dans cette section nous décrivons succinctement le procédé VUP pour *modéliser statiquement avec VUML*. Une description plus détaillée du VUP est donnée dans l'annexe B.

Comme nous l'avons introduit, VUML est un profil UML permettant de modéliser un système selon différents points de vues à l'aide de la notion de classe multivue. Un modèle VUML est donc un modèle UML qui peut contenir des classes multivue.

VUP propose une démarche en trois étapes principales pour élaborer un modèle VUML :

- **Analyser des Exigences** (*Requirement Analysis*) : cette étape consiste à analyser le système, c'est-à-dire à élaborer le modèle des cas d'utilisation UML.
- **Modéliser par point de vue** (*Create Single-view Analysis Models*) : cette étape consiste à établir des modèles de conception partiels selon chaque point de vue (acteur). Pour chaque acteur, il s'agit de modéliser en détail les cas d'utilisation dont il est acteur principal, puis de réaliser un digramme de classes UML (*Analysis Model*) associé au point de vue correspondant.
- **Fusionner les modèles partiels et Élaborer le modèle multivue** (*Creat a Multiview Analysis Model*) : cette étape a pour but de produire le modèle de conception multivues (*Multiviews Analysis Model*) à partir des modèles (*Analysis Model*) produits précédemment

La Figure IV-4 montre une description semi-formelle du VUP.

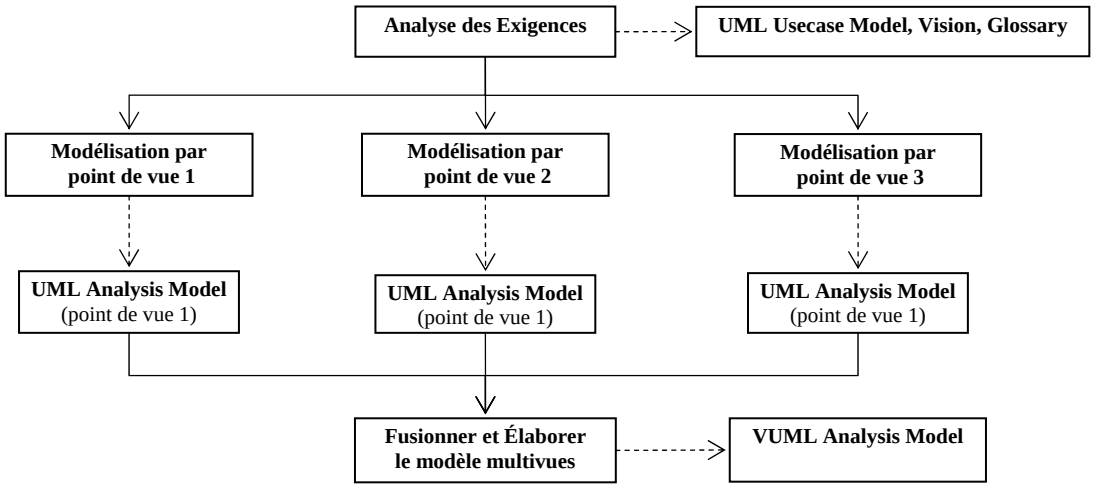


Figure IV-4. Noyau de la démarche de modélisation statique avec VUML

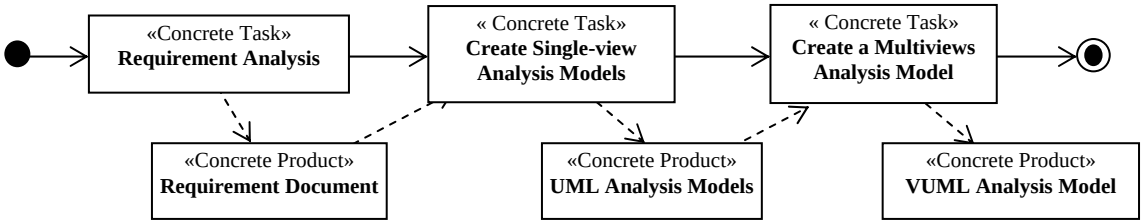
On peut constater qu'une grande partie de la démarche de VUP est une démarche de modélisation avec UML. Plus précisément, comparé avec le procédé RUP pour modéliser en UML, l'étape d'analyse des exigences et l'étape de modélisation par point de vue du VUP correspondent aux activités *Requirements* et *Usecase Analysis* du RUP (c.f. Tableau IV-1). Par conséquent, on peut construire le procédé VUP en réutilisant des activités du RUP. Les éléments du procédé VUP sont considérés comme concrets car ils concernent des produits concrets représentés en VUML.

Nous décrivons ci-dessous les principaux scénarios de la démarche de modélisation selon VUP. La mise en œuvre sous l'environnement PATPRO est illustrée dans la section **Erreur ! Source du renvoi introuvable.**.

- **Scénario 1 – Modélisation de la tâche racine du VUP**

Ce scénario illustre la situation où un modèle de procédé doit être élaboré «*from scratch*» s'il n'y a pas de patron convenable à réutiliser.

L'objectif de la tâche racine du VUP est d'analyser et concevoir un système. Elle reçoit en entrée une description informelle d'un système et élabore un modèle VUML du système. En cherchant dans la base de patrons (c.f. Tableau IV-1), nous ne trouvons pas de patron adressant le même problème ou ayant le même produit sortant[1] que la tâche racine du VUP. Nous définissons alors manuellement cette tâche pour obtenir le procédé décrit dans la Figure IV-5.



[1] S'il s'agit d'un patron décrivant une tâche de développement, en général, les conditions décrites dans le contexte résultant du patron reflètent ses produits sortants

Figure IV-5. Modèle de la tâche racine du VUP

- **Scénario 2 – Modélisation de la tâche *Requirements Analysis* du VUP en réutilisant des patrons**

Ce scénario illustre la définition d'un élément de procédé par réutilisation exacte (sans modification) de patrons.

L'objectif de la tâche *Requirements Analysis* du VUP est d'analyser des exigences afin d'élaborer un dossier contenant la vision du système, le glossaire et le modèle des cas d'utilisation du système. Parmi les patrons de la base (c.f. Tableau IV-1), il y a deux patrons adressant un problème d'analyse d'exigences : le patron général *Requirement Elicitation* et le patron concret *RUP Requirements*. Comme VUP modélise les exigences selon UML, le patron concret *RUP Requirements* est choisi comme patron le plus adapté à la spécification de la tâche *Requirements Analysis*. De plus, c'est une réutilisation exacte car à cette étape, les procédés VUP et RUP sont indifférenciés. De la même façon, on peut réutiliser le patron *RUP Requirement Document* pour définir le contenu du produit *Requirement Document* du VUP.

- **Scénario 3 – Modélisation de la tâche *Create Single-view Analysis Models* du VUP en réutilisant des patrons**

Ce scénario illustre la définition d'un élément de procédé par réutilisation avec adaptation de patrons.

La tâche *Create Single-view Analysis Models* du VUP a pour but d'élaborer plusieurs modèles d'analyse en UML selon les points de vue correspondant aux différents acteurs du système. Elle doit donc comporter une sous-tâche permettant d'élaborer un modèle d'analyse UML. Cette sous-tâche, appelée *Create a Single-view Analysis Model*, se répète pour chaque acteur de la liste d'acteurs extraite du dossier des exigences VUP.

Pour réaliser cette itération, on peut réutiliser le patron *Working with List* (c.f. Tableau IV-1) pour générer le contenu de la tâche *Create Single-view Analysis Models*. Cependant une telle réutilisation demande une adaptation car le patron *Working with List* est abstrait. Nous montrons dans les sections IV.3 et IV.4 la réalisation de ce scénario et le dépliage du modèle créé.

III. GESTION DES PATRONS DE PROCÉDÉ DANS PATPRO-MOD

Dans cette section nous illustrons l'utilisation de l'environnement PATPRO-MOD pour gérer la base de patrons de procédé.

La Figure IV-6 montre le menu affichant les fonctionnalités du module.

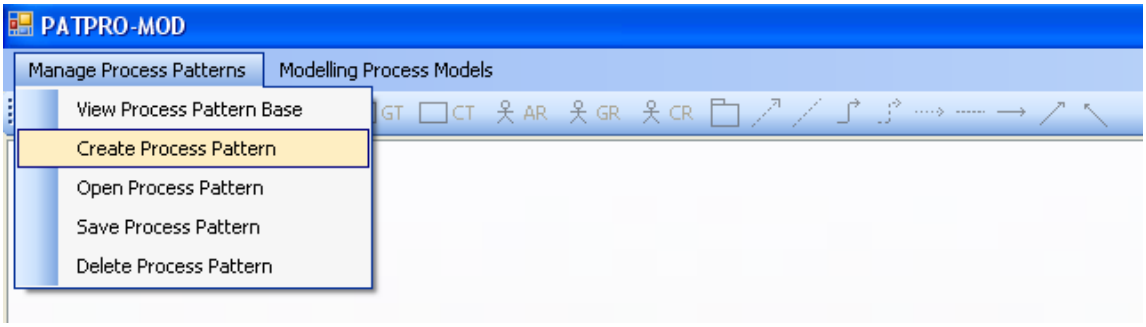


Figure IV-6. Fonctionnalités du module *Gestion de Patrons de Procédé*

III.1. CRÉATION D'UN PATRON DE PROCÉDÉ

La création d'un patron est la fonction centrale du module. Il y a deux activités à faire pour créer un patron : d'abord le spécifier et ensuite éditer sa solution. La création et la modification d'un patron travaillent sur un formulaire affichant les caractéristiques du patron.

La Figure IV-7 montre l'interface de PATPRO-MOD pour la spécification du patron abstrait *Working with List*. Le concepteur de patrons remplit ce formulaire avec les caractéristiques du patron : le nom, le niveau d'abstraction, le problème (intention), le contexte et éventuellement les paramètres formels.

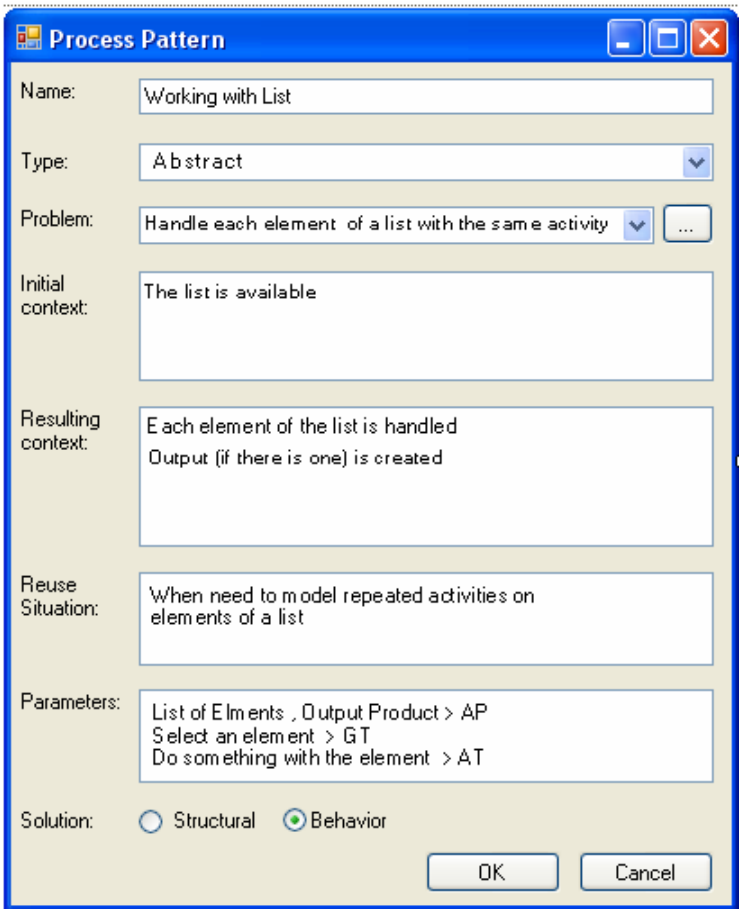


Figure IV-7. Caractéristiques du patron abstrait *Working with List*

Ensuite, selon l'option choisie dans la rubrique *Solution*, le concepteur peut continuer à éditer[1] un modèle (structurel ou comportemental) du procédé qui sera capturé dans la solution du patron. Par exemple, la Figure IV-8 montre un modèle comportemental créé pour la solution du patron *Working with List*. Finalement, le patron complet est montré dans la Figure IV-9.

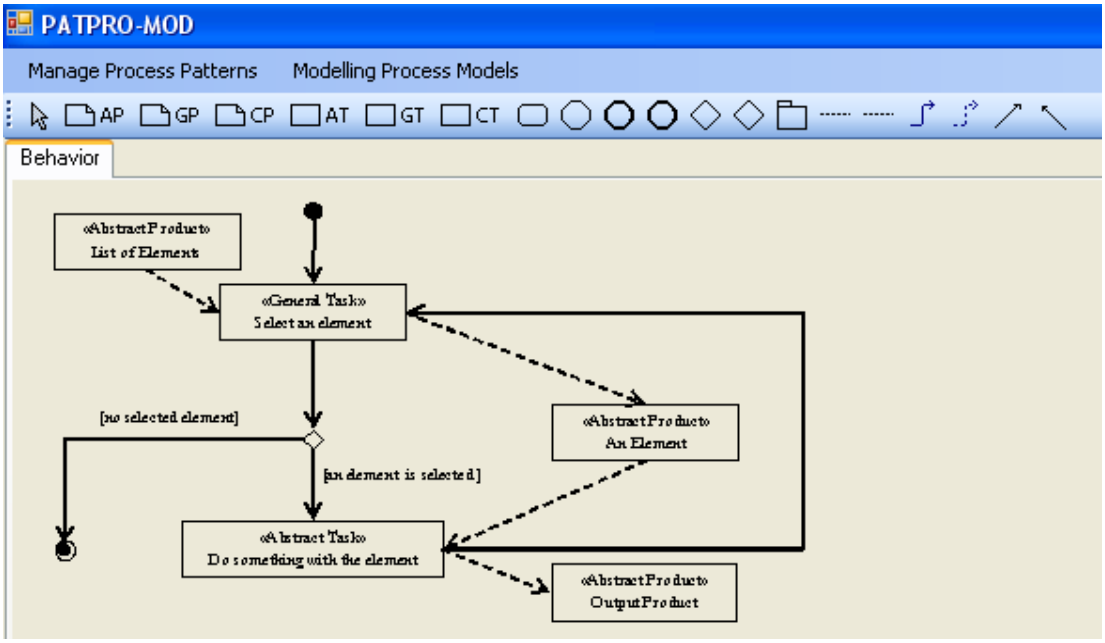


Figure IV-8. Création de la solution du patron *Working with List*

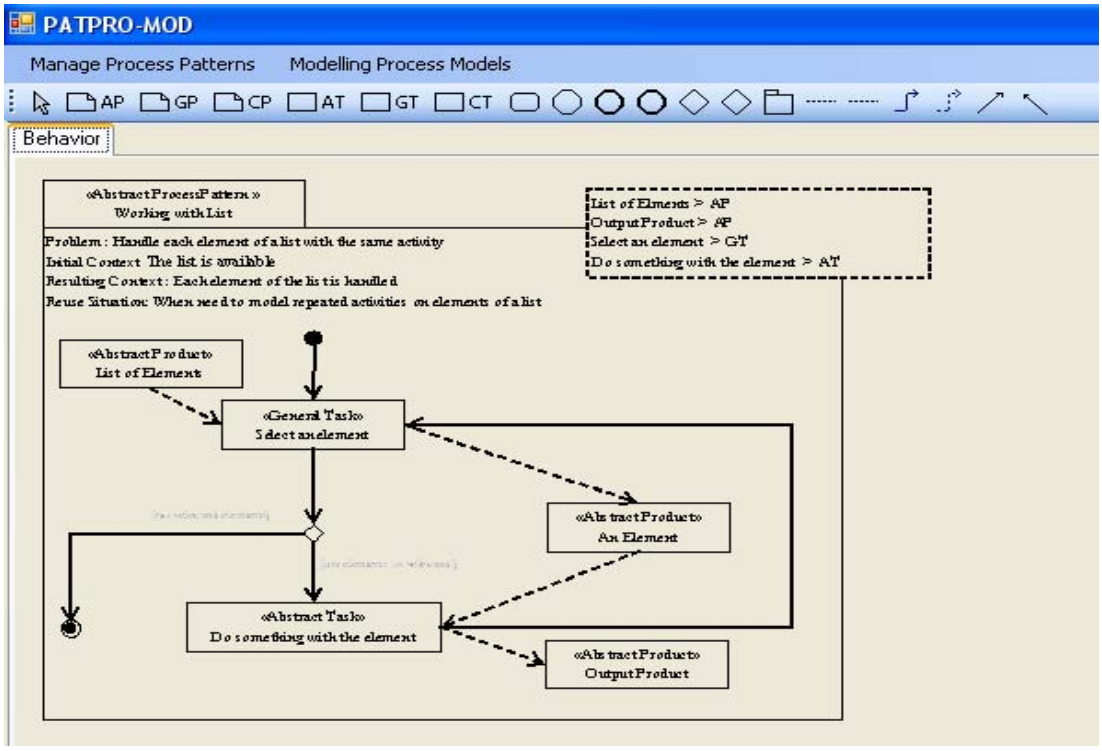


Figure IV-9. Modèle du patron abstrait *Working with list*

[1] Cette activité est réalisée en activant la fonction *Create Process Model* du module *Modélisation des Procédé*

III.2. MANIPULATION D'UN PATRON DE PROCÉDÉ

Dans un premier temps, nous ne fournissons qu'une interface simple pour la sélection des patrons à manipuler. L'utilisateur doit choisir manuellement un patron dans la liste de patrons et choisir ensuite l'opération à réaliser. Par exemple, la Figure IV-10 montre l'interface utilisateur correspondant à la liste des patrons du Tableau IV-1, avec sélection du patron *RUP Requirements*.

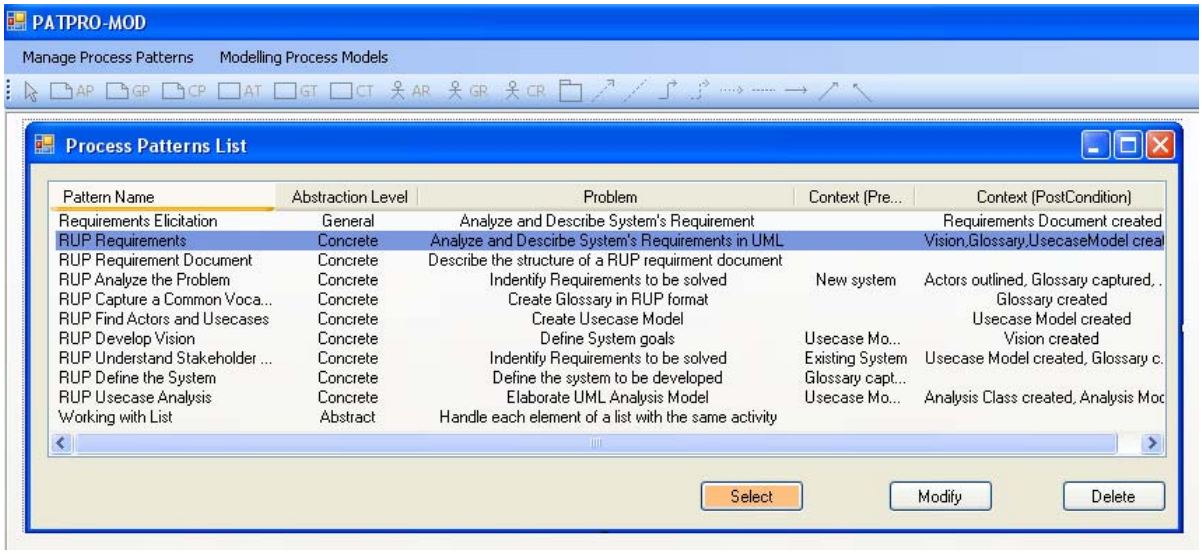


Figure IV-10. Interface pour la sélection d'un patron à manipuler

IV. MODÉLISATION DE PROCÉDÉS DANS PATPRO-MOD

Dans cette section nous illustrons l'utilisation du prototype PATPRO-MOD pour modéliser les procédés.

La Figure IV-11 montre l'interface du module avec le menu affichant ses fonctionnalités.

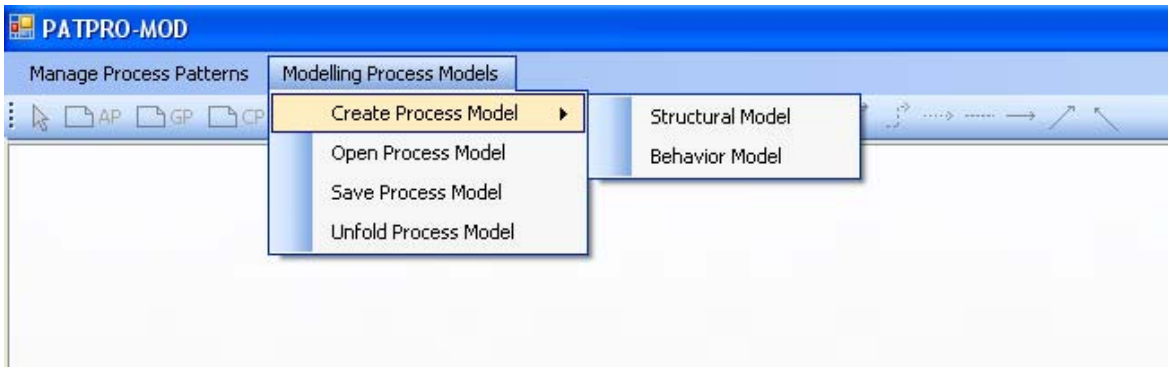


Figure IV-11. Fonctionnalités du module *Modélisation de procédés*

PATPRO-MOD permet de créer des modèles structurels ou comportementaux. Pour chaque type de modèle choisi, l'outil affiche des éléments de procédé et des relations valides pour le type de modèle.

Les Figure IV-12 et la Figure IV-13 montrent la barre d'outils correspondant à la fonction de création d'un modèle structurel ou comportemental.

Figure IV-12. Interface de la fonction de création d'un modèle structurel

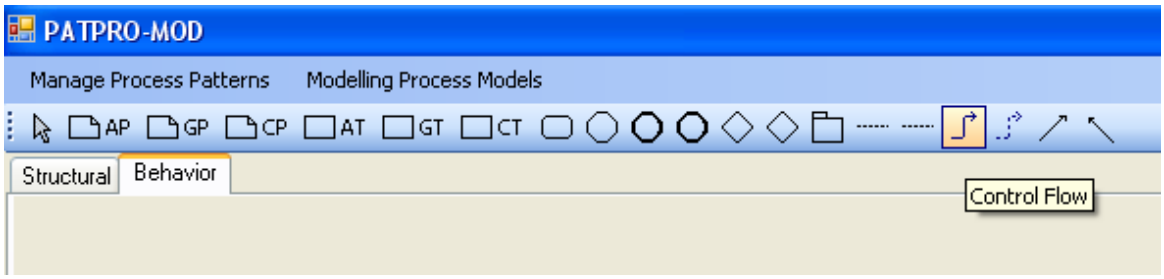


Figure IV-13. Interface de la fonction de création d'un modèle comportemental

Selon le type et le niveau d'abstraction d'un élément choisi, un formulaire approprié apparaît pour permettre au concepteur de spécifier l'élément. Par exemple, la Figure IV-14 montre le formulaire à remplir pour définir le produit concret *RUP Requirement Document*, et la Figure IV-15 celui de la tâche concrète *RUP Find Actors and Usecases*.

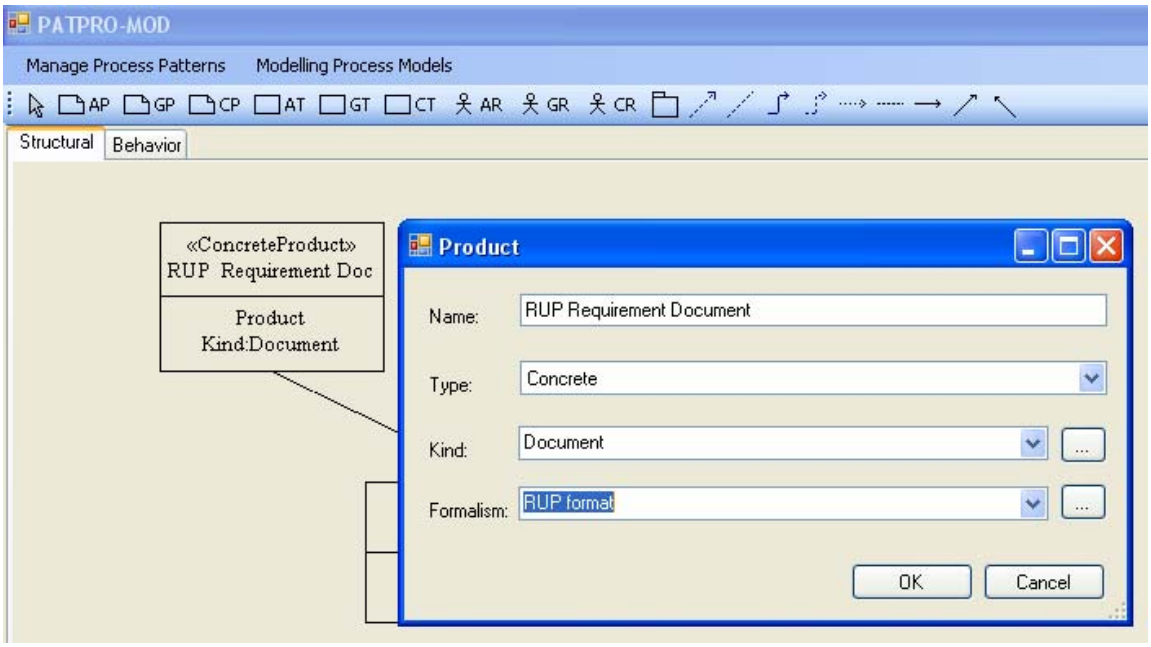


Figure IV-14. Définition du produit concret *RUP Requirement Document*

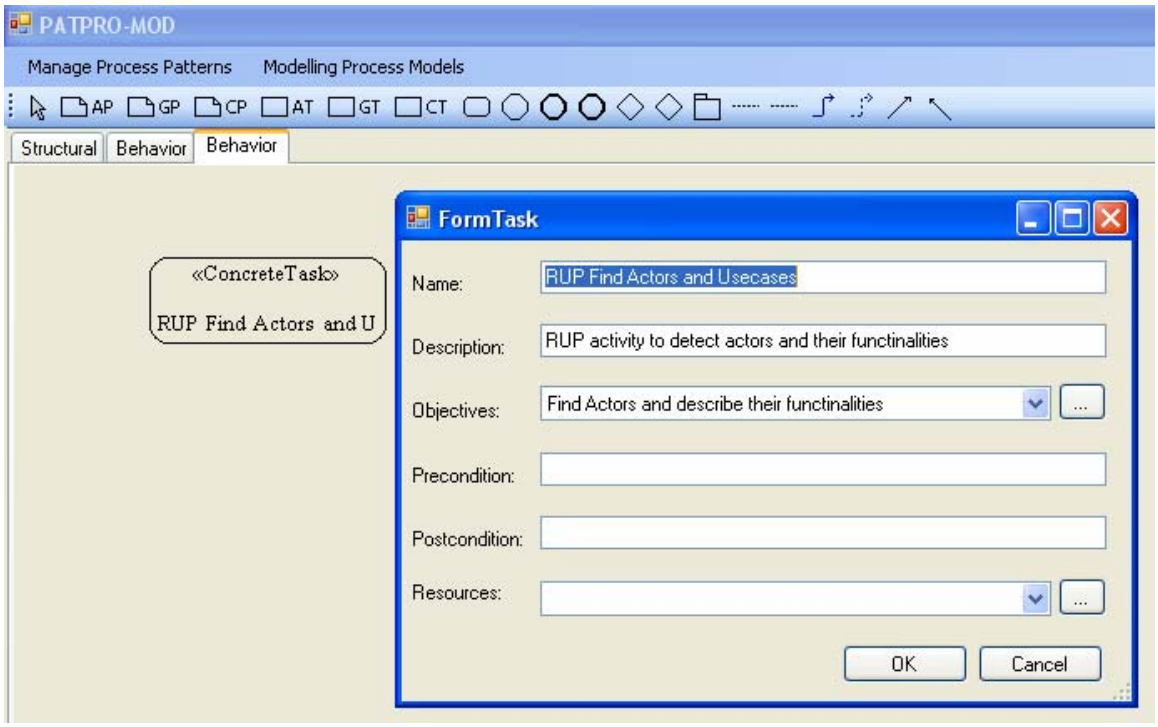


Figure IV-15. Définition de la tâche concrète *RUP Find Actors and Usecases*

Nous montrons dans la suite la modélisation du VUP en respectant les différents scénarios décrits ci-dessus dans la section II.2.

IV.1. MODÉLISATION D'UN PROCÉDÉ «FROM SCRATCH»

Comme la spécification de la tâche racine du VUP ne correspond à aucun patron de la base, il faut créer un modèle comportemental décrivant cette tâche (c'est-à-dire le procédé VUP lui-même). La Figure IV-16 montre le modèle créé.

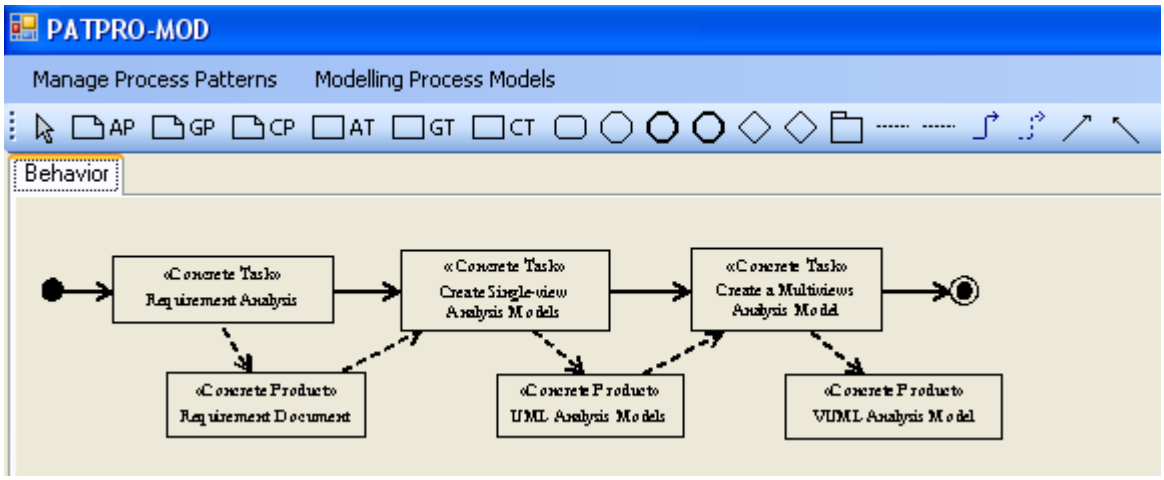


Figure IV-16. Modèle de la tâche racine du VUP

Ensuite, il faut raffiner les sous-tâches du VUP, en commençant par la tâche *Requirement Analysis*. Comme nous l'avons vu ci-avant, cette tâche peut être modélisée en réutilisant un patron approprié du RUP. La section suivante illustre une telle réutilisation.

IV.2. RÉUTILISATION EXACTE D'UN PATRON DE PROCÉDÉ

Pour modéliser la tâche *Requirement Analysis,* il faut d'abord chercher un patron adapté. Dans la version actuelle, l'outil ne supporte pas la recherche et la sélection automatique des patrons. Il ne fournit qu'un filtre simple pour sélectionner des patrons ayant des problèmes correspondant à l'objectif de la tâche. Le concepteur doit choisir lui-même un patron à appliquer dans la liste des patrons.

La Figure IV-17 montre l'interface de PATPRO-MOD avec l'icône de patron de procédé sur la barre d'outils de la fonction de création d'un modèle qui permet d'ouvrir la liste des patrons pour en choisir un à appliquer.

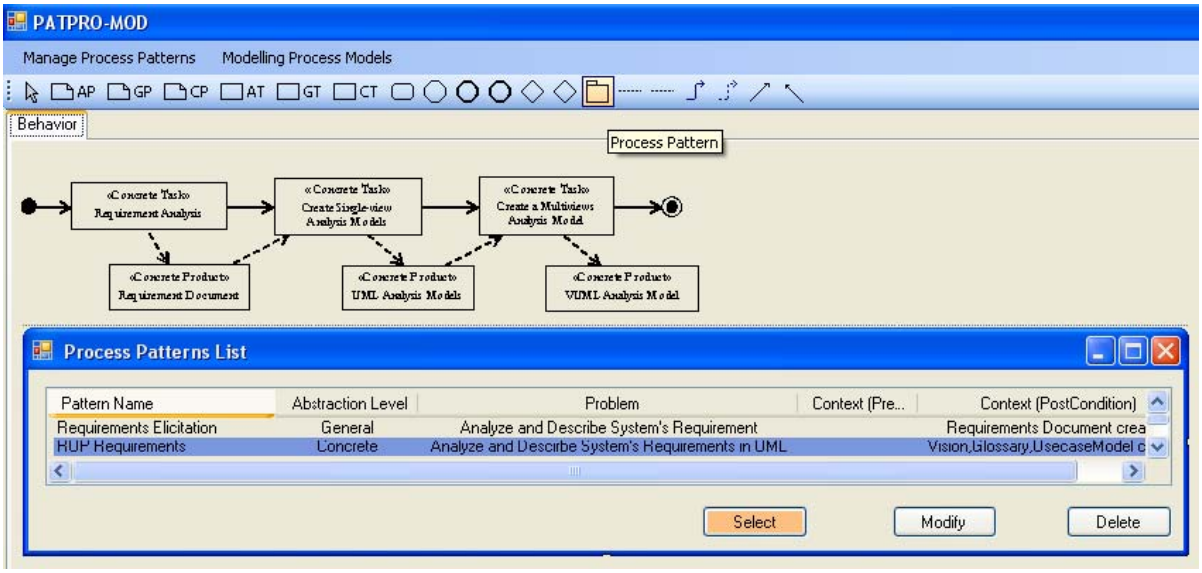


Figure IV-17. Sélection du patron *RUP Requirements* pour définir le contenu de la tâche *Requirement Analysis* du VUP

Après cette sélection, le patron concret *RUP Requirements* apparaît dans le modèle, et nous créons une relation *ProcessPatternBinding* entre la tâche *Requirement Analysis* et ce patron. Le patron concret *RUP Requirements* n'a aucun paramètre. Puisque nous voulons une réutilisation exacte de la solution de ce patron, il n'y a pas de paramètres à spécifier pour la relation *ProcessPatternBinding*.

Pour définir le produit concret *Requirement Document*, nous réutilisons également le patron concret *RUP Requirement Document*. La réutilisation est réalisée de la même manière que pour le patron *RUP Requirements*.

Nous montrons dans la Figure IV-18 et la Figure IV-19 le patron *RUP Requirements* et le patron *RUP Requirement Document*, respectivement. La figure Figure IV-20 montre le modèle VUP après la définition de la tâche *Requirement Analysis*.

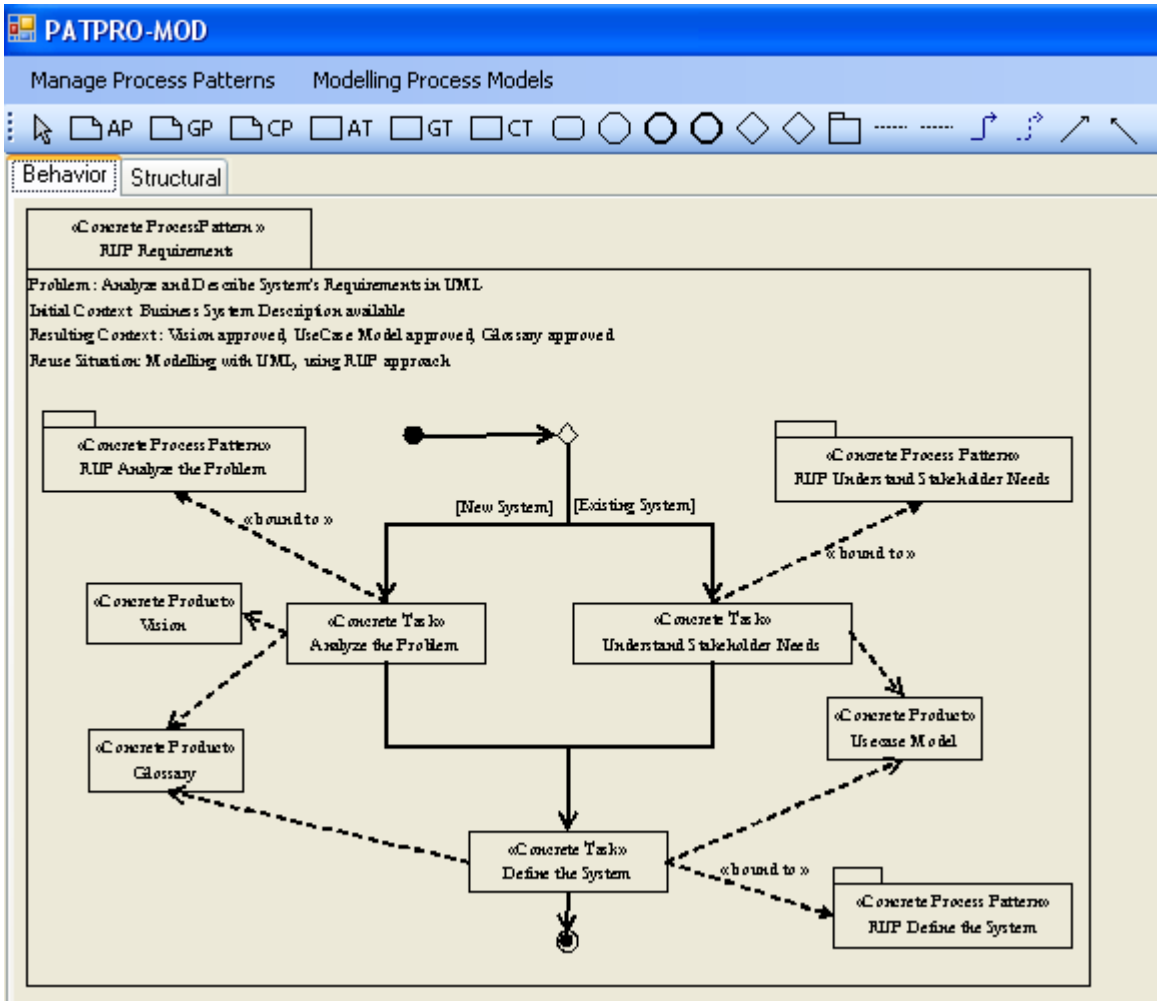


Figure IV-18. Le patron concret *RUP Requirements*. Dans la solution de ce patron, on peut voir les tâches basées sur d'autres patrons.

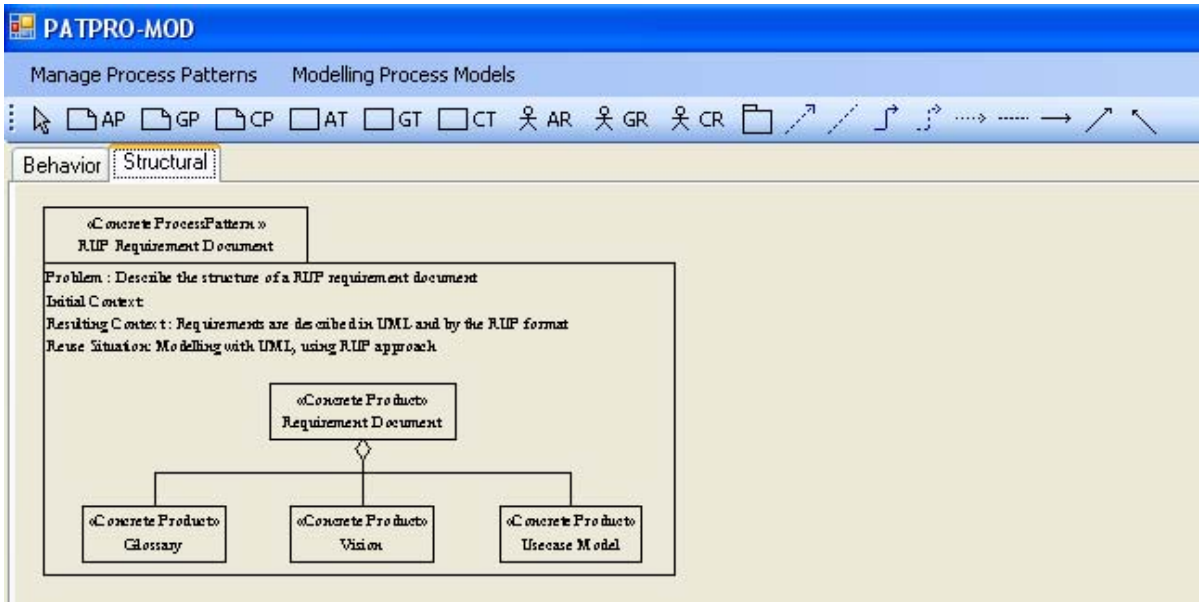


Figure IV-19. Le patron concret *RUP Requirement Document*

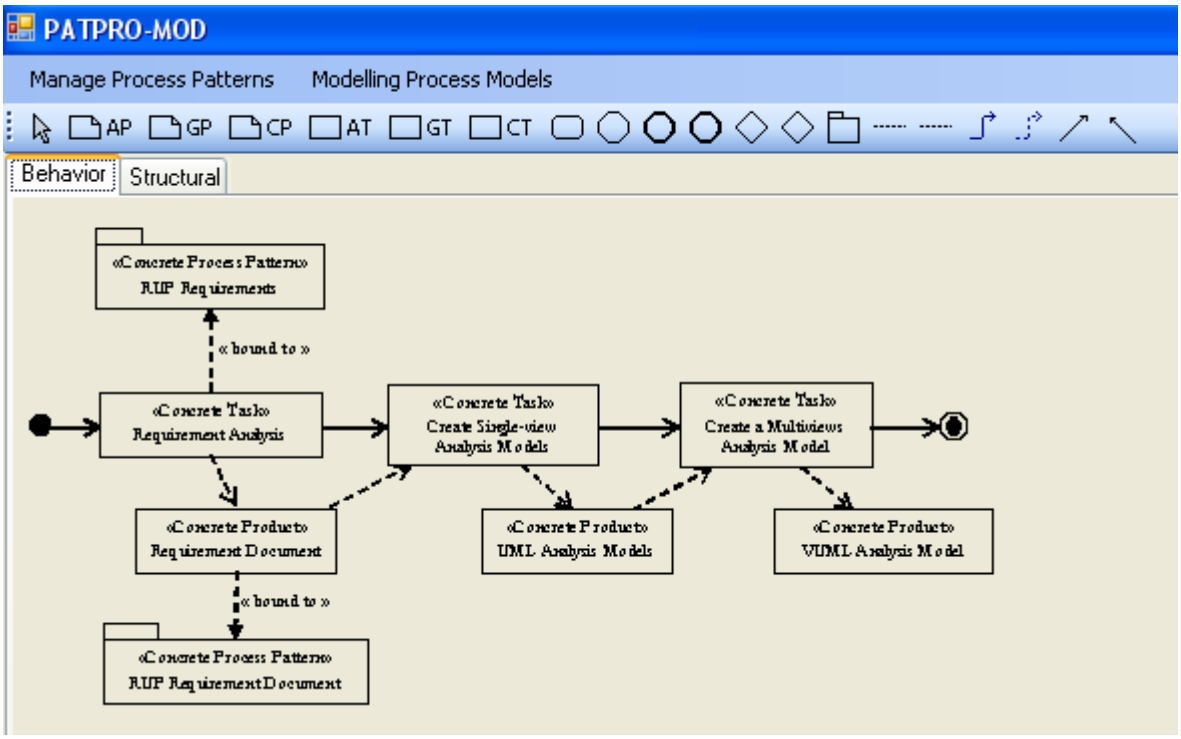


Figure IV-20. Définition de la tâche *Requirement Analysis* du VUP par réutilisation de type *binding* de deux patrons

IV.3. RÉUTILISATION AVEC ADAPTATION D'UN PATRON DE PROCÉDÉ

Nous décrivons maintenant la modélisation de la tâche *Create Single-view Analyis Models*. Comme nous l'avons expliqué ci-dessus, cette tâche est en fait une itération de la sous-tâche *Create a Single-view Analysis Model* qui élabore un modèle d'analyse UML selon un point de vue donné.

Pour générer le contenu de la tâche *Create Single-view Analyis Models* nous utilisons le patron *Working with List* (Figure IV-9) pour décrire une telle itération. Comme le niveau d'abstraction du patron *Working with List* est différent celui de la tâche *Create Single-view Analyis Models*, en établissant la relation *ProcessPatternBinding* entre eux, nous devons spécifier une substitution de paramètres pour adapter la solution du patron aux besoins de la tâche.

Dans cette application, le paramètre formel *List of elements* est substitué par la liste d'acteurs extraite du dossier des exigences ; le produit sortant est spécifié par le paramètre effectif *UML Analysis Model* ; la tâche *Select an element* est spécifiée par le paramètre effectif *Select an actor ;* la tâche *Do something with the element* est spécifiée par le paramètre effectif *Create a Single-view Analysis Model.* Les trois derniers paramètres effectifs sont déclarés avec le type StringExpression, ce qui signifie qu'ils sont générés en dépliant la relation *ProcessPatternBinding.*

La Figure IV-21 montre la définition de la tâche *Create Single-view Analyis Models* par réutilisation du patron *Working with List.*

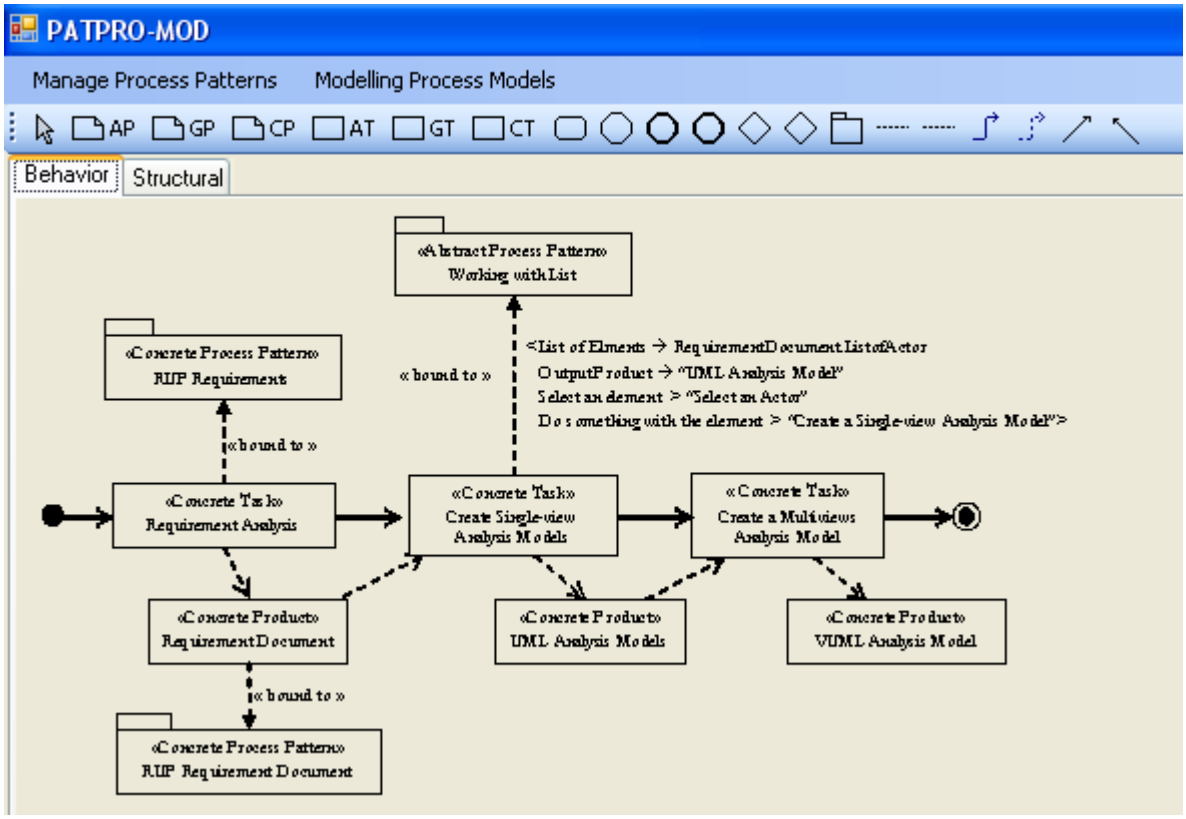


Figure IV-21. Définition de la tâche *Create Single-view Analysis Models* du VUP par réutilisation de type *binding* patron *Working with List* avec adaptation par substitution de paramètres.

IV.4. DÉPLIAGE D'UN MODÈLE BASÉ SUR DES PATRONS

Pour générer automatiquement le modèle résultant d'une relation d'application de patrons, PATPRO-MOD propose la fonction de dépliage d'un modèle.

Cette fonction permet de choisir, dans le modèle en cours, un élément ayant une relation avec un patron, et d'imiter le patron pour élaborer un nouveau modèle décrivant l'élément en question. La Figure IV-22 montre l'interface de cette fonction.

Dans cet exemple nous montrons le dépliage du modèle décrivant la tâche *Create Single-view Analysis Models* reliée au patron *Working with List* par une relation de type binding (cf. Figure IV-21). Après la sélection de cette tâche, PATPRO-MOD génère un modèle de procédé associé à la tâche *Create Single-view Analysis Models*. La Figure IV-23 montre le modèle généré par ce dépliage.

Après la génération du nouveau modèle de procédé, nous pouvons continuer à le raffiner pour obtenir un modèle détaillé. Par exemple, nous montrons dans la Figure IV-24 la définition de la tâche *Create a Single-view Analysis Model*. Cette tâche doit élaborer un modèle d'analyse selon le point de vue d'un acteur. Pour cela, elle itère sur les cas d'utilisation dans lesquels l'acteur joue le rôle principal. Pour traiter cette itération et définir le contenu de la tâche, nous réutilisons encore une fois le patron *Working with List*.

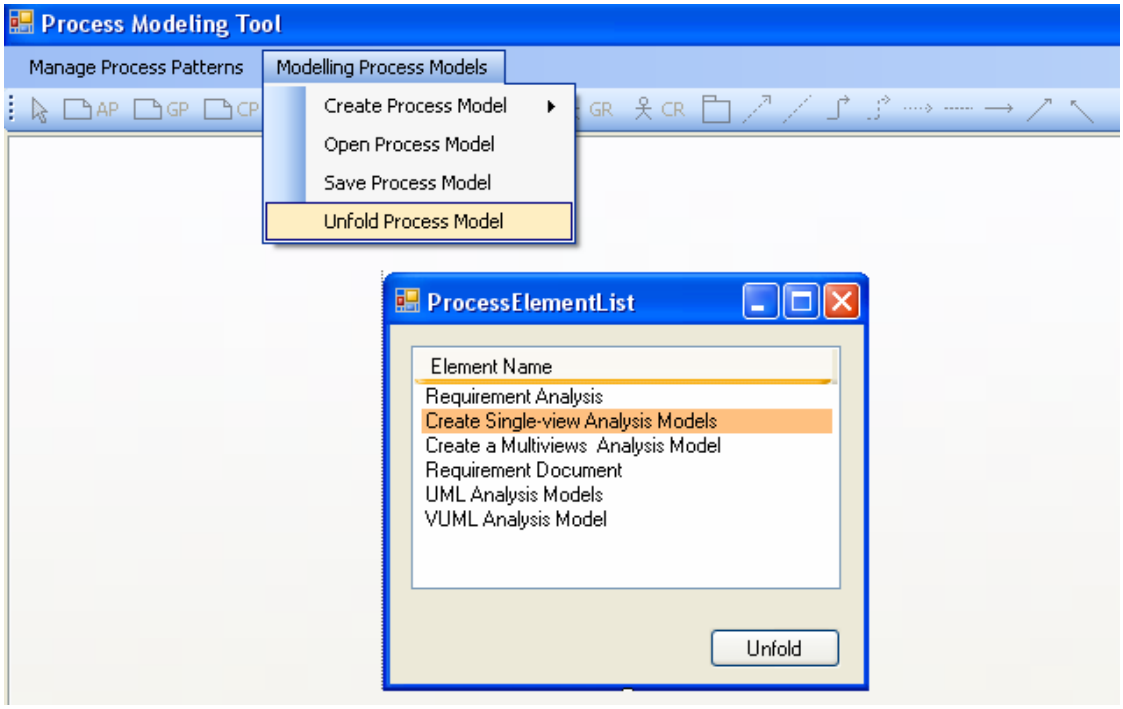


Figure IV-22. Interface de la fonction de dépliage d'un modèle

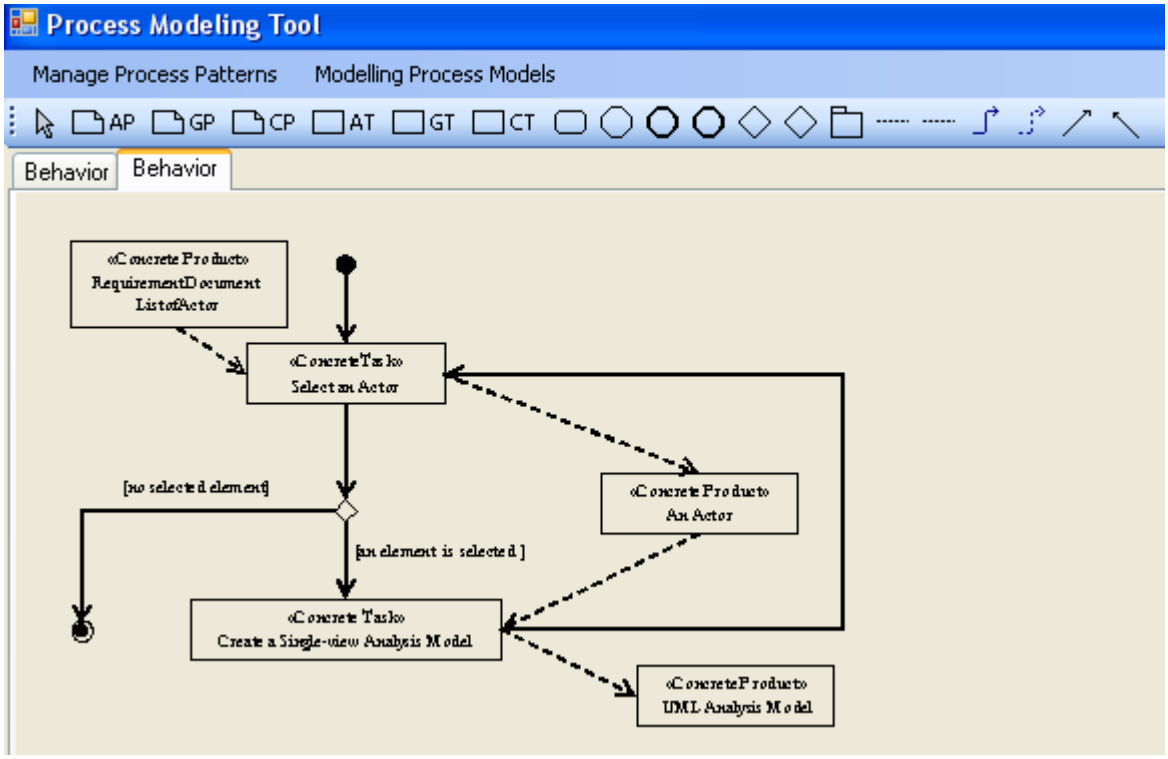


Figure IV-23. Résultat du dépliage du modèle décrivant la tâche *Create Single-view Analysis Models*

La Figure IV-25 montre le résultat du dépliage de la relation *ProcessPatternBinding* entre *Create a Single-view Analysis Model* et *Working with List*, et la réutilisation du patron *RUP Usecase Analysis* pour définir la tâche *Usecase Analysis* du VUP. Cette réutilisation est illustrée par une relation *ProcessPatternBinding*.

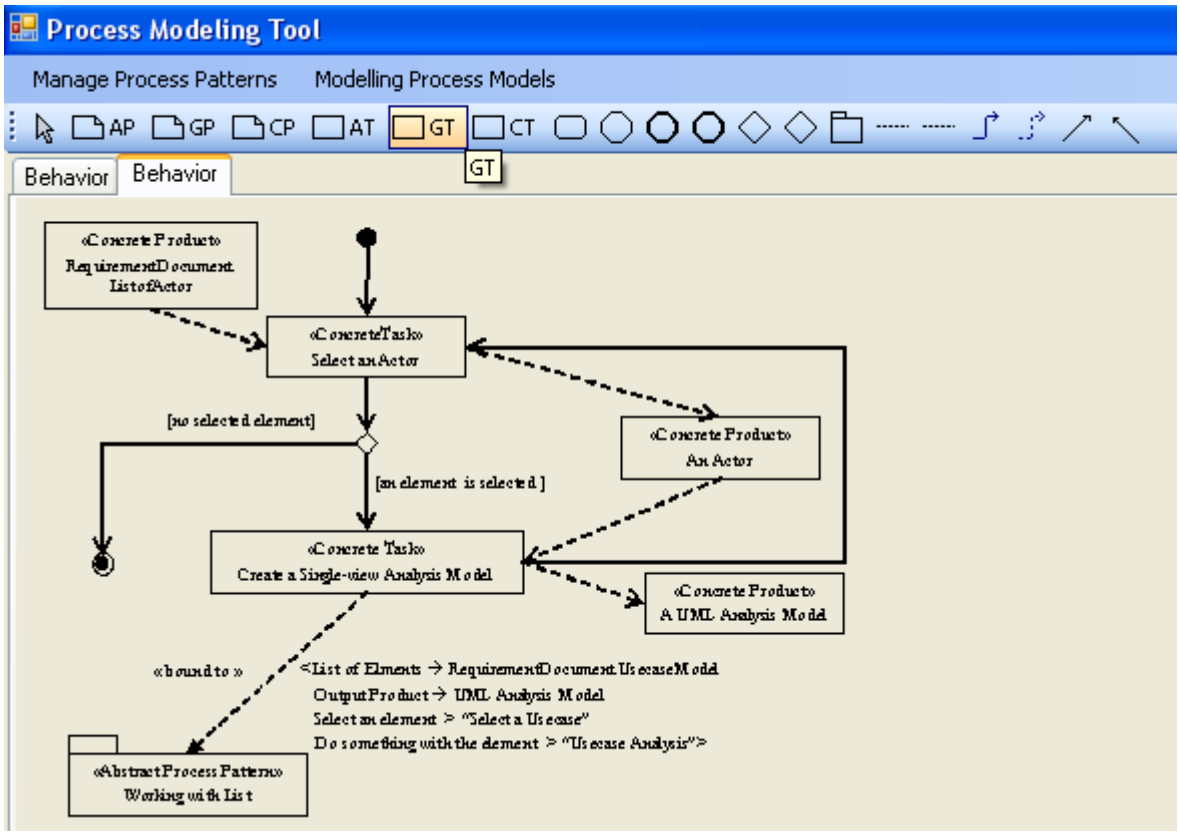


Figure IV-24. Définition de la tâche *Create a Single-view Analysis Model* du VUP en réutilisant le patron *Working with List* avec adaptation par substitution de paramètres.

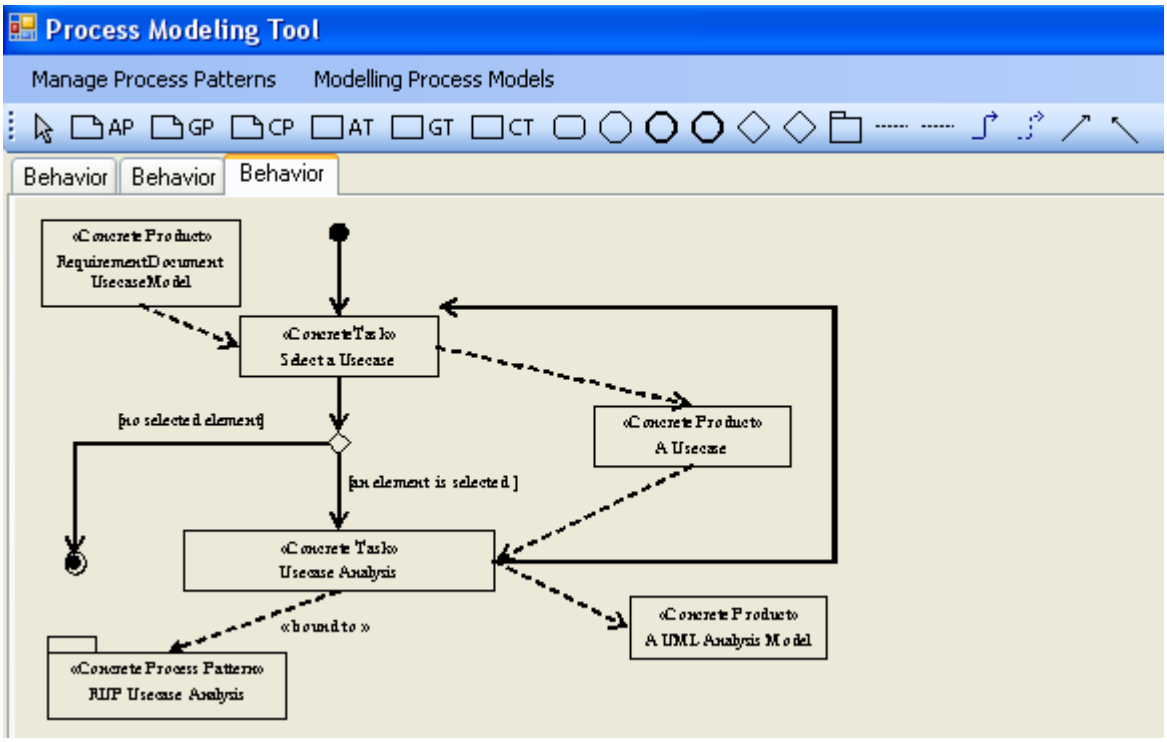


Figure IV-25. Résultat du dépliage du modèle de la tâche *Create a Single-view Analysis Model* et réutilisation par une relation *binding* du patron *RUP Usecase Analysis*

V. CONCLUSION

Dans ce chapitre nous avons décrit la réalisation informatique associée à ce travail de thèse. L'objectif était de valider les propositions théoriques et méthodologiques présentées dans les chapitres précédents. Plus précisément, nous avons décrit l'environnement PATPRO-MOD en illustrant son utilisation à travers l'étude de cas *Modélisation du procédé VUP*.

Les fonctionnalités de PATPRO-MOD ont été complètement analysées, mais au moment de l'impression de ce document, tout n'a pas été implémenté.

Le travail le plus important a été la réalisation de l'éditeur (graphique) supportant le méta-modèle UML-PP pour le module *Modélisation de procédés*. Nous avons implémenté et testé les fonctions de création, de modification des modèles de procédé basés sur des patrons réutilisables. La fonction de dépliage a été implémentée pour permettre une application simple mais automatique des patrons. Cependant, à ce jour, nous ne traitons pas les conflits ou incohérences pouvant survenir lors de l'application de patrons (c.f section III du Chapitre III). Nous sommes en train de développer et implémenter des algorithmes pour traiter de tels conflits. De même, PATPRO-MOD n'est pas en mesure de gérer la cohérence de l'ensemble des modèles de procédé décrivant un procédé. Dans l'état actuel, nous n'avons pas encore de base de données pour stocker les éléments de procédé et les modèles associés.

Le module *Gestion des Patrons de procédé* a été implémenté d'une façon simple, pour fournir un ensemble de patrons à réutiliser au cours de la modélisation. Il serait possible de développer une base de donnée pour stocker les patrons, et bien entendu, le raffinement de la fonction de recherche de patrons appropriés fait partie des perspectives à court terme de notre travail.

CONCLUSION GÉNÉRALE

Dans cette thèse, nous avons étudié le concept de patron de procédé dans le contexte du développement de logiciels. Nous avions pour objectif de formaliser ce concept pour permettre la représentation de procédés à bases de patrons réutilisable, et de proposer une méthode pour permettre la réutilisation systématique de patrons en modélisant les procédés.

Contributions

L'originalité de notre travail est de considérer le concept de patron de procédé avec une vision large pour d'une part capturer divers types de connaissances sur les procédés, et d'autre part proposer des moyens pour réutiliser de façon automatique ou semi-automatique ces patrons dans la modélisation de procédés.

Plus précisément, les résultats suivants peuvent être mis à l'actif de notre travail de recherche :

(a) une synthèse des problématiques liées à la réutilisation de procédés

Nous avons fait une large étude bibliographique sur les travaux existant dans le domaine de la modélisation et de la réutilisation de procédés. Cette étude nous a permis de faire une synthèse des problèmes de réutilisation de procédés et de proposer un cadre de référence pour évaluer les travaux dans cet axe.

(b) une clarification et un élargissement du concept de patron de procédé

Nous avons également réalisé une analyse des travaux relatifs aux patrons de procédé et établit un l'état de l'art sur ce sujet [TranHN05c]. Cet analyse nous a permis de proposer une nouvelle définition du concept de patron de procédé [TranHN05b][TranHN07a] ainsi qu'une classification des patrons de procédé [TranHN05a].

(c) un méta-modèle de procédé intégrant le concept de patron de procédé

Nous avons développé le méta-modèle UML-PP [TranHN06a][TranHN07b] pour formaliser le concept de patron de procédé et la manière d'appliquer les patrons dans la modélisation de procédés.

Pour prendre en compte l'objectif de compréhensibilité, notre méta-modèle est inspiré de SPEM1.1, et conforme au MOF. Dans le méta-modèle UML-PP, nous avons défini une syntaxe abstraite ainsi qu'une syntaxe concrète pour un LDP basé sur UML et dédié au domaine des procédés, incluant celui des patrons de procédé.

Plus précisément, UML-PP définit les éléments spécifiques de procédés et de patrons de procédé qui ne sont pas supportés par le méta-modèle UML. Ces éléments permettent d'une

part la représentation de patrons de procédé dans les modèles de procédé, d'autre part la description et l'organisation de patrons de procédé.

UML-PP permet de décrire la structure interne d'un patron de procédé (*problème, solution, contexte*) ainsi que les relations entre les patrons (*utilisation, raffinement, variance*). Nous caractérisons en particulier les patrons de procédé par leur niveau d'abstraction (*abstrait, général, concret*) afin de couvrir la diversité des connaissances en procédé, en allant du plus générique au plus spécifique.

Pour représenter explicitement l'application de patrons dans la phase de modélisation de procédés, nous définissons les patrons de procédé comme des éléments paramétrables, et proposons deux relations *ProcessPatternBinding* et *ProcessPatternApplying* exprimant l'utilisation de patrons de procédé pour respectivement *définir un élément de procédé* et *(re)organiser un groupe d'éléments d'un modèle de procédé*.

Pour renforcer la sémantique statique du méta-modèle et assurer la cohérence des modèles conformes à UML-PP, nous avons défini un ensemble de règles de bonne modélisation exprimées en langage OCL.

(d) un méta-procédé pour guider l'application des patrons de procédé d'une façon systématique et automatisable

Sur le plan méthodologique, nous proposons une démarche de modélisation de procédés fondée largement sur la réutilisation de patrons de procédé [TranHN06b][TranHN06c]. Cette démarche permet d'élaborer progressivement un modèle de procédé représenté en UML-PP. Elle est décrite sous forme d'un méta-procédé à grain fin décrit lui-même en UML-PP afin de fournir aux concepteurs de procédé un guidage précis et faciliter la mise en œuvre dans des outils supports.

Pour permettre une automatisation de l'application de patrons de procédé, nous avons défini des opérateurs de réutilisation de patrons qui réalisent certaines tâches du méta-procédé. Nous fournissons ainsi les opérateurs *ProcessPatternSearching* et *ProcessPatternSelecting* pour aider les concepteurs à choisir des patrons à réutiliser, les opérateurs *ProcessPatternBinding* et *ProcessPatternApplying* pour générer les résultats de l'application de patrons en dépliant les relations établies entre les patrons et les éléments de procédé dans un modèle. Une sémantique opérationnelle décrite avec le langage de méta-programmation Kermeta a été définie pour ces opérateurs, en particulier les opérateurs de recherche et d'imitation.

(e) un environnement supportant la modélisation par réutilisation de patrons de procédé

Pour concrétiser nos propositions, nous avons réalisé le prototype PATPRO-MOD qui permet l'application automatique de patrons en modélisant les procédés logiciels. PATPRO-MOD a été développé en *C#* sous l'IDE *Sharpdevelop* pour la plateforme Windows. Il fournit deux fonctions principales :

- créer et gérer des catalogues de patrons de procédé,

- élaborer des modèles de procédé en UML-PP en réutilisant (semi)automatiquement des patrons stockés dans un catalogue de patrons.

L'environnement implémente en fait le méta-procédé PATPRO pour guider de façon prescriptive les concepteurs de procédé. Concrètement, les activités de modélisation de procédés réalisées dans PATPRO-MOD sont à réaliser dans l'ordre décrit par le méta-procédé. Au moment de la rédaction de cette thèse, le prototype aide à générer automatiquement le modèle résultant de l'application d'un patron, mais la sélection du patron à appliquer est réalisée manuellement par le concepteur.

Une expérimentation a été faite en utilisant cet outil pour modéliser le procédé VUP, qui est le procédé d'élaboration de modèles multivue associé au profil VUML. Cela nous a permis d'une part de valider nos propositions, d'autre part de constituer les bases d'une plateforme de gestion de patrons de procédé.

Perspectives

Le travail effectué dans cette thèse peut être poursuivi dans plusieurs directions.

Sur le plan théorique, nous envisageons d'approfondir les points suivants :

- **Formaliser les notions de problème et de contexte d'un patron de procédé**

Une des suites logiques de notre travail porte sur la formalisation des notions de *problème* et de *contexte* d'un patron. En effet, ces notions sont des aspects décisifs, lors de la réutilisation, pour permettre de choisir, parmi un ensemble de patrons, celui qui est le plus adapté à un besoin particulier. Il apparaît donc intéressant et nécessaire de formaliser de façon plus précise ces notions pour d'une part mieux organiser les catalogues des patrons, et d'autre part améliorer les opérateurs de recherche de patrons.

- **Identifier d'autres types d'application des patrons de procédé**

Nous avons mis en évidence deux types d'application des patrons de procédé pour générer le contenu d'un élément de procédé (*ProcessPatternBinding*), et pour restructurer ou enrichir en groupe d'éléments de procédé (*ProcessPatternApplying*). Cependant il peut être intéressant d'étendre cette typologie en identifiant d'autres types d'application de patrons de procédé, par exemple l'application de patrons pour guider la composition de modèles de procédé.

- **Approfondir la sémantique des opérateurs d'imitation de patrons**

Nous avons défini les opérateurs d'imitation (*ProcessPatternBinding.Unfold()* et *ProcessPatternApplying.Unfold()*) pour déplier les modèles basés sur les relations *ProcessPatternBinding* et *ProcessPatternApplying*. Une sémantique opérationnelle des opérateurs associés a été définie. Cependant, pour compléter cette sémantique, il est nécessaire d'approfondir les points suivants :

- Relation *ProcessPatternBinding* : pour l'instant, nous ne permettons qu'une seule relation *ProcessPatternBinding* établie entre un élément de procédé et un patron source. Nous envisageons une modification au niveau méta-modèle pour permettre de représenter

plusieurs relations « binding » entre un élément et différents patrons qui peuvent jouer le rôle de patron source. Cela peut permettre de représenter différents schémas de réalisation d'un élément. Dans ce cas, il faut ajouter des contraintes et modifier la sémantique de l'opérateur associé pour assurer notamment qu'une seule relation sera dépliée à l'exécution (de l'opérateur).

– Relation *ProcessPatternApplying* : il reste à creuser les sujets suivants :
  - *La complétude des modes d'application* : nous avons proposé trois modes d'application de la relation *ProcessPatternApplying*. Il faut examiner s'il peut y avoir d'autres modes d'application.
  - *Les conflits se produisant en appliquant l'opérateur ProcessPatternApplying* : Nous avons discuté de certains conflits potentiels relatifs à l'application de patrons par la relation *ProcessPatternApplying*. Cependant, la liste de conflits proposée n'est pas complète, et les algorithmes pour traiter ces conflits n'ont pas été totalement développés. Par ailleurs, la résolution des conflits en appliquant des patrons peut nécessiter l'intervention du concepteur ou l'application d'heuristiques, ce qui sort du cadre initial de cette thèse.
  - *L'application de différents patrons à un même groupe d'éléments de procédé* : rien n'interdit de réutiliser plusieurs patrons simultanément pour restructurer ou enrichir un groupe d'éléments de procédé. Mais les règles concernant ce scénario n'ont pas été définies.

- **Exploiter les relations d'organisation de procédé**

Nous avons défini des relations d'organisation (*PatternUse, PatternVariation, PatternRefinement*) de patrons. Nous avons prévu de les exploiter pour améliorer les algorithmes de recherche et de sélection de patrons.

- **Aligner/intégrer le travail avec les autres LDP basés sur UML et/ou SPEM**

En développant le méta-modèle UML-PP, nous avons mis l'accent sur la définition et la représentation du concept de patron de procédé. Notre objectif à moyen terme est d'aligner ou intégrer notre méta-modèle avec un méta-modèle de procédé normatif supportant la représentation et éventuellement l'exécution de procédés.

Dans cette optique, nous considérons plusieurs possibilités. Nous pourrions aligner notre méta-modèle avec SPEM 2.0 lorsqu'il sera officiellement adopté. Toutefois, cette orientation est un peu aléatoire car SPEM 2.0 semble bien compliqué pour être adopté largement par la communauté. Une autre piste envisageable est d'intégrer notre travail à un LDP basé sur UML et/ou SPEM, qui pourrait être moins général que SPEM mais plus simple à utiliser. Nous pensons par exemple à UML4SPM (*UML for Software Process Modeling*), un LDP exécutable développé au laboratoire LIP6 [Bendraou05]. En effet, UML4SPM s'appuie sur UML en prenant en compte les concepts de SPEM, et est supporté par un environnement d'exécution. Cependant, UML4SPM n'intègre pas explicitement le concept de patron de procédé. Il serait donc intéressant d'étudier l'intégration de notre approche par réutilisation de patrons dans UML4SPM.

Sur le plan de l'implémentation, nous envisageons les travaux suivants :

- **Améliorer l'environnement PATPRO-MOD**

L'outil PATPRO-MOD est actuellement un prototype exploratoire destiné à être étendu par itérations successives. Nous voulons raffiner et compléter l'implémentation de ses fonctionnalités pour avoir un véritable environnement de modélisation de procédés supportant la réutilisation de patrons de procédé.

- **Élaborer des bases de patrons de procédé**

Bien entendu, pour appliquer efficacement notre approche dans des projets réels, nous avons besoin d'identifier et de collecter des patrons de procédé à différents niveaux d'abstraction, et dans différents domaines d'application. Une typologie des problèmes, des contextes est également nécessaire pour aider à organiser les patrons en bases de patrons.

BIBLIOGRAPHIE

[ACuna01] Acuña, S.T., Ferré X., "Software Process Modelling". In *Proceedings of the World Multiconference on Systemics, Cybernetics and Informatics SCI'01*, Orlando, USA, 2001.

[Alexander79] Alexander C., *The Timeless Way of Building*. New York: Oxford University Press, 1979

[Ambler98] Ambler S.W., *Process Patterns: Building Large-Scale Systems Using Object Technology*. New York: SIGS Books/Cambridge University Press, 1998.

[Ambler99] Ambler S.W., *More Process Patterns: Delivering Large-Scale Systems Using Object Technology*. New York: SIGS Books/Cambridge University Press, 1999.

[Ambriola97] Ambriola, V., Conradi, R., Fuggetta, A., "Assessing process-centered software engineering environments". *ACM Transactions on Software Engineering and Methodology* 6, 3 (1997) 283-328.

[APE02] APE Software Technik Praktikum 2002. APE – Applied Patterns Environment. 2002. http://www4.informatik.tu-muenchen.de/~ape

[APPPerfect02] AppPerfect Code Analyzer. http://www.appperfect.com/products/codeanalyzer.html

[Arbaoui96] Arbaoui, S.,Oquendo, F., "Reuse sensitive process models: are process elements softwareassets too?". In *Proceedings of the International Software Process Workshop ISPW'96*, Dijon, France, 1996.

[Armenise93] Armenise, P., Bandinelli, S., Ghezzi, C., and Morzenti, A., "Survey and assesment the process representing formalisms". *GOODSTEP Technical Report No. 015*, December 1993.

[Avrilionis95] Avrilionis, D., Cunin, P.Y., "Evolution handling in PSEEs" In *Proceedings of the Software Engineering and its Applications GL'95*, Paris, 1995.

[Avrilionis96] Avrilionis, D., Cunin, P.Y., Fernström, C., "OPSIS: A View Mechanism for Software Processes which Supports their Evolution and Reuse". In *Proceedings of the International Conference on Software Enginnering ICSE'96*, Berlin, Germany, 1996.

[Baldi94] Baldi, M., Gai, S., Jaccheri, M.L., and Lago, P., "Object-Oriented Software Process Model Design in E3", In *[Finkelstein94]*, pp. 279-290.,1994.

[Bandinelli94] S. Bandinelli, A. Fuggetta, C. Ghezzi and L. Lavazza, "SPADE: An Environment for Software Process Analysis, Design, and Enactment", In *[Finkelstein94]*, pp. 223-247.,1994.

[Basili91] Basili, V.R., and Rombach, H.D., "Support for comprehensive reuse". *IEEE SoftwareEngineering Journal,* 6(5):303–316, September (1991)

[Basili94] Basili, V.R., Caldiera, G. and Rombach, H.D.,"Experience Factory," In *Encyclopedia of Software Engineering (John J. Marciniak, ed.)*, vol. 1, pp. 469--476, John Wiley Sons, 1994.

[Beck87] Beck, K., Cunnimgham, W., "Using pattern languages for object-oriented programs", In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications OOPSLA'87*, Orlando, USA,1987.

[Belkhatir94] Belkhatir, N., Estublier J.and Melo, W., "ADELE-TEMPO: An Environment to Support Process Modelling and Enaction", In *[Finkelstein94]* pp. 187-222.,1994.

[Benali92] Benali, K., Derniame, J. C., "Software processes modeling: what, who, and when".In *Proceedings of the European Workshop on Software Process Technology EWSPT'92,* Trondheim, Norway, 1992.

[Bendraou05] Bendraou R., Gervais M.P. and Blanc X., "UML4SPM: A UML2.0-Based metamodel for Software Process Modeling", In *Proceedings of the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems MoDELS'05*, Montego Bay, Jamaica, 2005.

[Bergner98] Bergner K., Rausch, A., Sihling M., "A Component Methodology based on Process Patterns". In *Proceedings of the Annual Conference on the Pattern Languages of Programs PLOP'98*, Monticello, Illinois, 1998

[Bernstein03] Bernstein P.A., Pottinger. R. A. "Merging Models Based on Given Correspondences". In *Proceedings of the International Conference on Very Large Databases VLDB'03*, Berlin, Germany, 2003.

[Birk97] Birk, A., Giese, P., Kempkens, R., Rombach, D., Ruhe G.(Editors). "The PERFECT Handbook (Vol. 2)". *Fraunhofer IESE Reports Nr. 060.97*, 1997.

[Boehm88] Boehm, B., "A spiral model for software development and enhancement", *IEEE Computer*, pages 61–72, 1988.

[Bruynooghe94] Bruynooghe, R. F., Greenwood, R. M., Robertson, I. Sa J. and Warboys, B. C., "PADM: Towards a Total Process Modelling System", In *[Finkelstein94]*, pp. 293-334. 1994.

[Buschmann96] Buschmann F., Meunier R., Rohnert et al., *Pattern-Oriented Software Architecture - A System of Patterns*, John Wiley, 1996

[Coad95] Coad P., North D. et Mayfield M., *Object Models – Strategies, Patterns and Application*, Yourdon Press Computing Series, 1995.

[Chou02] Chou,S.C., "A Process Modeling Language Consisting of High Level UML-based Diagrams and Low Level Process Language", *Journal of Object Technology*, vol. 1, no. 4, September-October 2002, pages 137-163.

[CMMI02] *Capability Maturity Model® Integration* (CMMISM), Version 1.1, CMU/SEI, 2002

[Conte01] Conte A., Fredj M. Giraudin JP., Rieu D., "P-Sigma : un formalisme pour une représentation unifiée de patrons", In *Acte du XIXème Congrès Informatique des Organisations et Systèmes d'Information et de Décision Inforsid01*, Genève, 2001

[Conte02] Conte A., Fredj M., Hassine I., Giraudin J-P., Rieu D., "AGAP : an environment to design and apply patterns", In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics SMC02*, Tunisia, 2002.

[Conradi91] Conradi, R., Liu, C., and Jaccheri, M. "Process modeling paradigms: an evaluation". In *Proceedings of International Software Process Workshop ISPW'91*, San Francisco,USA,1991.

[Conradi94a] Conradi, R. et al., "EPOS: Object-Oriented and Cooperative Process Modeling", In*[Finkelstein94]*, p.362 , 1994.

[Conradi94b] R. Conradi, C. Fernström, A. Fuggetta, "Concepts for evolving software processes". In In*[Finkelstein94]*, p.9-31, 1994.

[Conradi97] Conradi, R., Liu, C., "Revised PMLs and PSEEs for Industrial SPI". In *Proceedings of the European Conference on Object-Oriented Programming ECOOP'97* , Jyväskylä, Finland, 1997.

[Coplien92] Coplien J.O., *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[Coplien94] Coplien J.O, "A Development Process Generative Pattern Language", In *Proceedings of the Annual Conference on the Pattern Languages of Programs PLoP'94*, 1994.

[Coplien96] Coplien J.O., *Software Patterns*. SIGS Book & Multimedia, 1996.

[Coulette00] Coulette B., Crégut X., Dong T. B. T. and Tran D. T., "RHODES, a Process Component Centered Software Engineering Environment", In *Proceedings of the International Conference on Enterprise Information Systems ICEIS'00*, Stafford, UK, 2000.

[Coulette01] Coulette B., Crégut X., Dong T. B. T. and Tran D. T., "Managing process through a base of reusable components", In *Proceedings of the International Conference on Enterprise Information Systems ICEIS'01*, Setubal, Portugal, 2001

[Coulette02] Coulette B., Crégut X., Dong T. B. T. and Tran D. T., "A Metaprocess to define and reuse process components", In *Proceedings of the International Conference on Integrated Design and Process Technology IDPT'02*, Pasadena, USA, 2002.

[Crégut97] Crégut X. et Coulette B. "PBOOL: an Object-Oriented Language for Definition and Reuse of Enactable Processes". *Software Concepts and Tools*, volume 18, numéro 2. Novembre 1997.

[CSharp06] SharpDevelop IDE, http://www.icsharpcode.net/OpenSource/SD/

[Curtis92] Curtis, B., Kellner, M., Over, J., "Process modeling". *Communications of the ACM 35*, 9 (September 1992) 75-90

[Dami98] Dami, S., Estublier, J. and Amiour, M., "APEL: A Graphical Yet Executable Formalism for Process Modeling". *Automated Software Engineering*, Vol. 5, No. 1, January 1998, pp. 61-96.

[Deiters90] W. Deiters and V. Gruhn, "Managing Software Processes in the Environment MELMAC", In *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environment,* pp. 193-205. 1990.

[Demeyer02] Demeyer S., Ducasse S., Nierstrasz O., *Object-Oriented Reengineering Patterns*, Morgan Kaufman Publishers, 2002

[Deneckere01] Deneckere R., Souveyet C., "Organising and selecting patterns in pattern languages with Process Maps", *In Proceedings of the International Conference on Object-Oriented Information Systems OOIS'01*, Calgary, Canada, 2001.

[Derniame99] Derniame J. C., Kaba B.A., Wastell D.(Editors), *Software Process: Principles, Methodology and Technology*. Lecture Notes in Computer Science 1500, Springer,1999

[Derniame04] Derniame J.C, and Oquendo F., "Key Issues and New Challenges in Software Process Technology", *UPGRADE, European Journal for the Informatics Professional*, Vol V, October 2004.

[Dewayne96] Dewayne E. P., "Practical Issues in Process Reuse", In *[ISPW96]* 1996.

[DiNitto02] Di Nitto, E., Lavazza, L., Schiavoni, M., Tracanella, E., Trombetta, M., "Deriving executable process descriptions from UML". In *Proceedings of the International Conference on Software Enginnering ICSE'02, Orlando, Florida, USA*, 2002.

[Dittmann02] Dittmann T., Gruhn V., Hagen M., "Improved Support for the definition and usage of process patterns". In *[SDPP02]* 2002.

[Dowson91] Dowson, M., Nejmeh, B., and Riddle, W. (1991). Fundamental software process concepts. In *Proceedings of European Workshop on Software Process Technology EWSPT'01,* Milan, Italy. 1991.

[EPF06] Eclipse Process Framework (www.eclipse.org/epf/).

[Fagan86] Fagan M.E., "Advances in Software Inspections", *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 7, Page 744-751, 1986

[Feiler93] Feiler, P. H., Humphrey, W. S., "Software process development and enactment: Concepts and definitions". In *Proceedings of the International Conference on Software Process ICSP 93*, Berlin, Germany, 1993.

[Finkelstein94] Finkelstein, A., Kramer, J., Nuseibeh, B.(Editors), *Software Process Modelling and Technology*. (Research Studies Press (Wiley), 1994.

[Firesmith01] Firesmith, D., Henderson-Sellers, B., *The OPEN Process Framework. An Introduction*, Addison-Wesley, December 2001, ISBN 0-201-67510-2

[Foote95] Foote, B. and Opdyke, W.F., "Life cycle and Refactoring Patterns That Support Evolution and Reuse". In *Proceedings of the Annual Conference on the Pattern Languages of Programs PLoP'95*, pp. 239-257, 1995.

[Fowler97] Fowler, M., *Analysis Patterns*, *Reusable Object Models,* Addison-Wesley, 1997

[Franch99] Franch, X. and Ribó, JM., "Some Reflexions in The Modeling of Software Process", In *Proceedings of the International Process Technology Workshop IPTW'99*, Villard de Lans, France, 1999.

[Fugetta96] Fuggetta, A., Wolf, A., *Software Process,* John Wiley & Sons, 1996

[Fugini92] Fugini, M. G., Pernici, B., "Specification Reuse in Conceptual Modeling, Database, Case, an integrated view of Information Systems Development", In *Proceedings of the Conference on Advanced Information Systems Engineering CAiSE'92*, Manchester, UK, 1992.

[Gamma94] Gamma E., Helm R., Johnson R., et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994

[Gnatz02] Gnatz M. Marschall F., Popp G., Rausch A., Schwerin W., "Common Meta-Model for a Living Software Development Processes", In *[SDPP02],* 2002.

[Gnatz03] Gnatz M. Marschall F., Popp G., Rausch A., Schwerin W., "The Living Software Development Process", *Journal Software Quality Professional*, Volume 5, Issue 3, June 2003

[Godart99] Godart C., Molli P., Perrin, O., "Modeling and enacting processes: Some difficulties". In *Proceedings of the International Process Techgnology Workshop IPTW99*, Villard de Lans, France,1999.

[Gomaa00] Gomaa, H., Kerschberg, L., Farrukh, G., "Domain Modeling of Software Process Models", In *Proceedings of the IEEE International Conference on the Engineering of Complex Computer Systems*, Tokyo, Japan, 2000.

[Groenewegen96] Groenewegen, L., Engels, G., "Reuse of software process fragments is reuse of software too", In *[ISPW96]*, 1996.

[Gzara00] Gzara L., *Les patterns pour l'ingénierie des Systèmes d'Information Produit*, Doctorat de l'INPG, spécialité Génie Industriel, Décembre 2000.

[Gzara03] Gzara L., Rieu D., Tollenaere M., "Product Information Systems Engineering : An Approach For Building Product Models By Reuse Of Patterns", *Robotics and Computer-Integrated Manufacturing*, volume 19, issue 3, june 2003, pages 239-261

[Hagen04] Hagen M., GruhnV., "Process Patterns - a Means to Describe Processes in a Flexible Way". *International Workshop on Software Process Simulation and Modeling ProSim'04*, Edinburgh, United Kingdom, 2004

[Harrison96] Harrison N.B., "Organizational Patterns for Teams". In *Proceedings of the Annual Conference on the Pattern Languages of Programs PLoP'96*, pp. 345-352, 1996.

[Heiman96] P. Heiman, G. Joeris, C. A. Krapp and B. Westfechtel, "DYNAMITE: Dynamic Task Nets for Software Process Management", In *Proceedings of the International Conference on Software Engineering ICSE'96*, Berlin, Germany, 1996.

[Henninger98] S. Henninger. An Environment for Reusing Software Processes. In *Proceedings of the 5th IEEE International Conference on Software Reuse (ICSR5)*, Victoria, BC, Canada, 1998.

[Hollenbach96] Hollenbach, C., Frakes, W.: Software Process Reuse in an Industrial Setting, In *Proceeding of the Fourth International Conference on Software Reuse*, Orlando, USA,1996.

[Huff88] K. E. Huff and V. Lesser, "A Plan-Based Intelligent Assistant that Supports the Software Development Process", In *Proceedings of the 3rd ACM Symposium on Practical Software Development Environments.* , pp. 97-106, ACM Press 1988.

[Huff96] K. Huff, "*Software process modeling*". In Software Process. (John Wiley & Sons, 1996)

[Humphrey88] Humphrey, W., "Characterizing the software process: a maturity framework". *IEEESoftware*, pages 73–79, 1988.

[IDEF093] National Institute of Standards and Technology, Integration Definition for Function Modelling (IDEF0), US Department of Commerce, Technology Administration, Federal Information Processing Standards Publication, Report Number FIPS PUB 183, 1993.

[IEEE97] IEEE Std 1074-1997, IEEE Standard for Developing Software Life Cycle Processes, 1997

[Iida99] Iida H., "Pattern-Oriented Approach to Software Process Evolution", In *Proceedings of the International Workshop on the Principles of Software Evolution IWPSE99*, Fuoka, Japan,1999.

[Iida02] Iida H., Tanaka Y., "A Compositional Process Pattern Framework for Component-based Process Modeling Assistance", In *[SDPP02]*, 2002.

[ISO95] ISO/IEC 12207, Information Technology Software Life Cycle Processes, 1995

[ISO98] ISO/IEC TR 15504-2. "Information Technology – Software process assessment – A reference model for processes and process capability", 1998.

[ISPW96] B. Boehm, editor. *Proceedings of the 10th International Software Process Workshop*, Dijon, France, June 1996. IEEE Computer Society Press.

[Jacobson99] Jacobson I., Booch G., Rumbaugh J., *The Unified Software Development Process*, Addisson-Wesley Object Technology Series, 1999

[Jørgensen00] Jørgensen, H. D., "Software Process Model Reuse and Learning", In Proceedings of the *Process Support for Distributed Team-based Software Development Workshop PDTSD'00,* Orlando, Florida, 2000.

[Junkermann94] G. Junkermann, B. Peuschel, W. Schafer and S. Wolf, "MERLIN: Supporting Cooperation in Software Development Through a Knowledge-Based Environment", In *[Finkelstein94]* , pp. 103-129,1994.

[Kaiser88] G. Kaiser, P. H. Feiler and S. S. Popovich, "Intelligent Assistance for Software Development and Maintenance". *IEEE Software*, No. 5, May 1988, pp. 40-49.

[Katayama89] Katayama, T., "A hierarchical and functional software process description and its enaction". In *Proceedings of the International Conference on Software Engineering ICSE89*, pages 343–352, Pittsburgh, Pennsylvania, 1989.

[Kawalek94] Kawalek P., and Wastell, DG., "The Development of a Process Modelling Method", In *Information systems methodologies, proceedings of the 2nd BCS conference on information systems methodologies, Heriott-Watt University*, 1994.

[Kellner91] Kellner, M., "Multiple-paradigms:approach for software process modeling". In *Proceedings of the International Software Process Workshop ISPW91*, San Francisco, CA, 1991.

[Kellner96] Kellner, M., "Reuse of Process Elements". In *Proceedings of the International Software Process Workshop ISPW96*,France, 1996.

[Kerth95] Kerth N., "Caterpillar's Fate: A Pattern Language for Transformation from Analysis to Design", In *Proceedings of the Annual Conference on the Pattern Languages of Programs PLoP95*, 1995.

[Kruchten03] Kruchten P., *The Rational Unified Process: An Introduction, Third Edition*. Addison Wesley. 2003

[Kurtev06] Kurtev I, Didonet Del Fabro M, "A DSL for Definition of Model Composition Operators", *Second Workshop on Models and Aspects - Handling Crosscutting Concerns in MDSD (ECOOP 2006)*, 2006

[Lindquist97] Lindquist, T. and Derniame, J.C., "Towards Distributed and Composable Process Components", In *Proceedings of European Workshop on Software Process Technology EWSPT07*, 1997.

[LISA02] LiSa - A Living Software Development Process Support Tool. 2002. http://processpatterns.informatik.tu-muenchen.de

[Lonchamp93] Lonchamp, J., "A structured conceptual and terminological framework for software process engineering". Proceedings of the Second International Conference on Software Process (February 1993) 41-53.

[Lonchamp98] Lonchamp J., "Process Model Patterns for Collaborative Work" 15th IFIP World Computer Congress, Telecoop'98 Vienne . 1998

[Madachy06] Madachy, RJ., Reusable Model Structures and Behaviors for Software Processes. SPW/ProSim2006, China, May 2006.

[Madhavji90] Madhavji N.H., et al., "Prism = Methodology + Process-oriented Environment," In *Proceedings of the International Conference on Software Engineering ICSE 1990*, Nice, 1990

[Malone03] Malone T.W, Crowston K., Herman G.A. (editors).. *Organizing Business Knowledge: The MIT Process Handbook* . Cambridge, MA: MIT Press, 2003.

[Mazz94] ESA-PSS-05

[MDA01] OMG, Model Driven Architecture, http://www.omg.org/mda/specs.htm, 2001

[Mili95] Mili, H., Mili, F., Mili, A., "Reusing Software: Issues and Research Directions", *IEEE Transactions on Software Engineering*, vol. 21, n°6, juin 1995

[MOF06] OMG, Meta Object Facility Core Specification version 2.0, http://www.omg.org/cgi-bin/doc?formal/2006-01-01, 2006

[Montangero94] Montangero, C. and Ambriola, V., "OIKOS: Constructing Process-Centred SDEs", In *[Finkelstein94]*, 1994, pp. 335-353.

[MSF] Microsoft Solutions Framework, Microsoft, http://www.microsoft.com/msf

[Nassar05] Nassar M., Guiochet J., Coulette B., Ebersold S., Crégut X. et Kriouile A.. « Vers un profil UML pour la conception de composants multivues ». *RSTI série l'OBJET*, volume 11, numéro 4 , pages 83-113. 2005

[Neu03] Neu H. and Rus I., "Reuse in Software Process Simulation Modeling", *In Proceedings of Software Process Simulation Modeling Workshop ProSim03* Portland, OR, 2003.

[Nguyen94] Nguyen M.N. and Conradi R., "Classification of Meta-processes and their Models". In *Proceedings of the International Conference on Software Engineering ICSE'94*, Washington, 1994.

[OCL05] MOG, OCL2.0 Specification, http://www.omg.org/cgi-bin/doc?formal/06-05-01, 2005

[OMII04] Sun Java coding conventions. www.omii.ac.uk/dissemination/JavaCodingStandards.pdf

[OPF] Open Process Framework Website http://www.donald-firesmith.com/

[Osterweil87] Osterweil, L. J., "Software processes are software too". In *Proceedings of the International Conference on Software Engineering ICSE'87*,Monterey, CA, 1987.

[Pavlov04] Pavlov V., Malenko, D. "Mining MSF for Process Patterns: a SPEM-based Approach", *VikingPLoP04*, Uppsala , 2004.

[Penker00] Penker, M., Eriksson, H.E., *Business Modeling With UML: Business Patterns at Work*, John Wiley & Sons, 2000.

[Pham08] Pham, M.T. "Un outil de modélisation de procédés supportant la réutilisation de patrons de procéddé". Mémoire de Master dirigé par Tran H.N., Université Nationale du Vietnam à HoChiMinh Ville, à soutenir en 2008.

[Priéto87] Priéto-Diaz, R., Freeman, P. "Classifying Software for Reusability", *IEEE Software*, vol. 4, n°1, January, 1987.

[Ralyté03] Ralyté, J., Deneckère, R., Rolland, C. "Towards a Generic Model for Situational Method Engineering", CAISE03

[Reddy06] Reddy Y. R., Ghosh S., France R., Straw G., Bieman J.M., McEachen N. , Song, E., Georg G. "Directives for Composing Aspect-Oriented Design Class Models". *T. Aspect-Oriented Software Development* I: 75-105 (2006)

[Ribo00] Ribó J.M., Franch X., "PROMENADE, a PML intended to enhance standardization, expressiveness and modularity in SPM". *Research Report LSI-00-34-R*, Dept. LSI, Politechnical University of Catalonia (2000).

[Ribo01] Ribó J.M., Franch X., "Building Expressive and Flexible Process Models using an UMLbased approach". In *Proceedings of the European Workshop in Software Process Technology EWSPT01*, Witten, Germany, 2001.

[Ribo02] Ribó J.M., Franch X., "Supporting Process Reuse in PROMENADE". *Research Report LSI-02-14-R*, Dept. LSI, Politechnical University of Catalonia (2002).

[Rieu99] Rieu, D., Giraudin, J.P., Saint Marcel, C., Front-Conte, A., "Des opérations et des relations pour les patrons de conception" , In *Acte du Congrès Informatique des Organisations et Systèmes d'Information et de Décision Inforsid'99*, La Garde, Var, France, 1999.

[Rolland98] Rolland,C., "A comprehensive view of Process Engineering", In *Proceedings of the Conference on Advanced Information Systems Engineering CAiSE98*, Italy, 1998.

[Royce70] Royce, W. W., "Managing the development of large software systems". *Proc IEEE WESCON*, 1970.

[Rupprecht00] Rupprecht, C., Funnger, M., Knublauch, H. and Rose, T., "Capture and Dissemination of Experience about the Construction of Engineering Processes", In *Proceedings of the 12th Conference on Advanced Information Systems Engineering CAISE 2000*, Stockholm, Sweden, 2000.

[Scacchi99] Scacchi, W., "Experience with software process simulation and modeling", *Journal of Systems and Software* 46 (1999) 183±192, 1999

[Scacchi00] Scacchi, W., "Understanding Software Process Redesign using Modeling, Analysis and Simulation", *Software Process: Improvement and Practice* 5(2-3): p.183-195, 2000.

[Schleicher98] Schleicher, A., Westfechtel, B. and Jäger, D. "Modeling Dynamic Software Processes with UML", *Technical Report AIB 98-11*, RWTH, Aachen, Germany, 1998.

[Schröder03] Schröder, J., The *Process Pattern Workbench*, Thesis (German), University Dortmund, 2003.

[SDPP02] Gnatz, M. Marschall, F., Popp, G., Rausch, A., Rodenberg-Ruiz, M., Schwerin, W (Editoirs), *Proceedings of the 1st Workshop on Software Development Process Patterns (SDPP'02)* at OOPSLA 2002, Seattle, Washington, USA, 2002.

[SFB501] SFB501:Development of large systems with generic methods, available at http://www.sfb501.uni-kl.de

[SPEM04] OMG, SPEM 2.0 RFP, http://www.omg.org/cgi-bin/doc?ad/2004-11-4, 2004.

[SPEM05] OMG, Software Process Engineering Metamodel v1.1 Specification, http://www.omg.org/cgi-bin/doc?formal/2005-01-06, 2005.

[SPEM07] OMG, SPEM 2.0 Draft Adopted Specification, http://www.omg.org/cgi-bin/doc?ptc/2007-02-01, 2007

[Storrle01] Storrle, H., "Describing Process Patterns with UML", In *Proceedings of the European Workshop in Software Process Technology EWSPT'01*, Witten, Germany, 2001

[Sutton95] S. Sutton Jr., D. Heimbigner and L. J. Osterweil, "APPL/A: A Language for Software Process Programming". *ACM Transaction on Software Engineering Methodology*, Vol. 4, No. 3, July 1995, pp. 221-286.

[Sutton97] Sutton, S.M., Osterweil, L.J., "The Design of a Next-Generation Process Language". In *Proceedings of Sixth European Software Engineering Conference*

together with Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering ESEC/FSE '97, 1997.

[TranDT01] Tran, D.T., *Formalisation et mise en oeuvre de la notion de composant de procédés logiciels*, Thèse doctorale, N°1818, Institut National Polytechnique de Toulouse, France, 2001.

[TranDT05] Tran, D.T, Tran, H.N., Coulette B., Crégut, X., Dong B.T., "Topological properties for characterizing well-formedness of Process Components". *Software Process: Improvement and Practice*, Wiley Interscience, V.10 N. 2, p. 217-247, May 2005.

[TranHN05a] Tran, H.N., Coulette B., Dong B.T., "A classification of Process Patterns", In *Proceedings of the International Conference on Software Development SWDC-REK'05*, Reykjavik, Island, 2005.

[TranHN05b] Tran, H.N., Coulette B., Dong B.T., "Towards a better understanding of Process Patterns", In *Proceedings of the International Conference on Software Engineering Research and Practice SERP 2005*, Las Vegas, USA., 2005

[TranHN05c] Tran, H.N., Coulette, B., "État de l'art sur les patrons de procédés", *Rapport IRIT*, *numéro 2005-11-R. 06,* Juin 2005.

[TranHN06a] Tran, H.N., Coulette B., Dong B.T., "A UML based process meta-model integrating a rigorous process patterns definition". In *Proceedings of the International Conference on Product Focused Software Process Improvement PROFES'06*, Amsterdam, Netherlands, 2006.

[TranHN06b] Tran, H.N., Coulette B., Dong B.T., "Exploiting Process Patterns for Software Process Models Reuse", In *Proceedings of the International Conference on Theories and Applications of Computer Science ICTACS'06*, HoChiMinh, Vietnam, 2006.

[TranHN06c] Tran, H.N., "Patterns-based Process Modelling and Reuse", represented at the *Doctoral Symposium of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems MoDELS'06*, Genova, Italy, 2006.

[TranHN07a] Tran, H.N., Coulette B., Dong B.T., "Broadening the Use of Process Patterns for Modelling Processes", In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering SEKE'07*, Boston, USA, 2007.

[TranHN07b] Tran, H.N., Coulette B., Dong B.T., "Modeling Process Patterns and Their Application", *International Conference on Software Engineering Advances ICSEA'07,*Cap Esterel, French Riviera, France, 2007.

[Triskell05] IRISA. Projet Triskell : KerMeta. http ://www.irisa.fr/triskell, 2005.

[Turgeon96] Turgeon, J., Madhavji, N.H.: "A Systematic, View-Based Approach to Eliciting Process Models". In *Proceedings of the European Workshop in Software Process Technology EWSPT96*, p.276-282, 1996.

[UML05a] OMG, UML 2.0 Infrastructure Specification, http://www.omg.org/cgi-bin/doc?formal/05-07-05, 2005

[UML05b] OMG, UML 2.0 Superstructure Specification, http://www.omg.org/cgi-bin/doc?formal/05-07-04, 2005

[v.d.Aalst03] van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B. and Barros, A.P., "Workflow Patterns", *Distributed and Parallel Databases*, 14(3), p. 5-51, 2003

[Vasconcelos98] Vasconcelos, F.M., de Werner, C.M.L.., "Organizing the Software DevelopmentProcess Knowledge: An Approach Based on Patterns"; *Journal of Software Engineering and Knowledge Engineering*, World Scientific Publishing Company (Vol. 8, n°. 4, 1998)

[Whitenack94] Whitenack, B., "RAPEL: A Requirements Analysis Pattern Language for Object-Oriented Development". *Proceedings of PLoP/94*

[Zamli01] K. Z. Zamli, Process Modeling Languages: A Literature Review, *Malaysian Journal of Computer Science*, Vol. 14 No. 2, December 2001, pp. 26-37

ANNEXE A.

IMPLÉMENTATION DES OPÉRATEURS DE RÉUTILISATION DE PATRONS DE PROCÉDÉ

Cette annexe fournit le code KerMeta [Triskell05] des opérateurs de réutilisation de patrons de procédé présentés dans la section II.2 du Chapitre III.

Nous avons utilisé le langage KerMeta pour décrire la sémantique opérationnelle des opérateurs de réutilisation car il nous permet de profiter à la fois de l'expressivité des langages de programmation et de la capacité d'exprimer les modifications de modèle avec des expressions OCL. Cependant pour simplifier, nous utilisons directement certaines opérations OCL en supposant qu'elles sont définies en tant que fonctions de KerMeta.

Opérateurs de Recherche

La Figure A montre l'extrait du méta-modèle UML-PP définissant les concepts relatifs aux patrons de procédé. Les opérateurs de recherche sont définis sous forme de méta-opérations de la méta-classe *ProcessPatternCatalogue*.

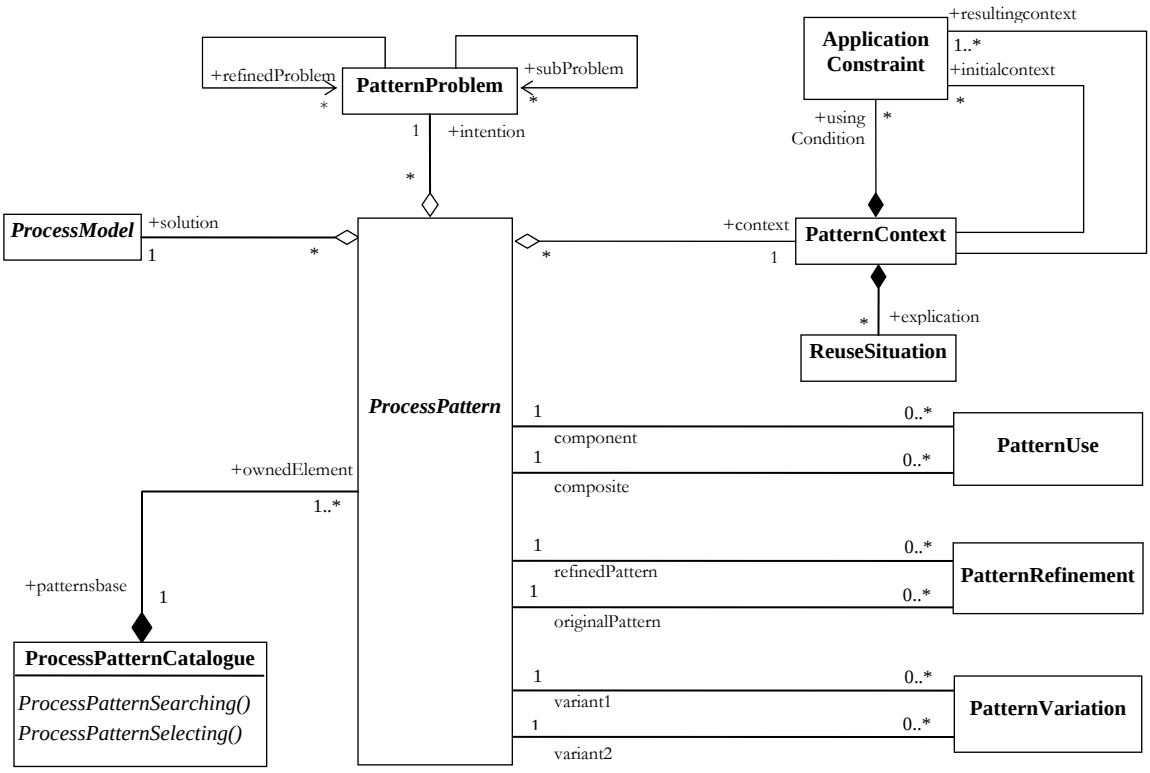


Figure A. Extrait du méta-modèle UML-PP décrivant l'organisation de patrons de procédé

Méta-Opération 1. ProcessPatternCatalogue.ProcessPatternSearching

```
class ProcessPatternCatalogue
{
operation ProcessPatternSearching(q :PatternProblem):set<ProcessPattern>
is do
// self.ownedElement est l'ensemble des patrons du catalogue
// p.intention est le problème du patron p
// p.intention.refined est l'ensemble des raffinements du problème du patron p
        result:= self.ownedElement.select{p|(p.intention == q}
                                    or (p.intention.refinedProblem.exist(i|i==q)}
end
}
```

Méta-Opération 2. ProcessPatternCatalogue.ProcessPatternSelecting

```
class ProcessPatternCatalogue
{
operation ProcessPatternSelecting(p: PatternProblem, c:PatternContext)
                                    :ProcessPattern
is do
// déclarer les variables temporaires
var convenientp : set<ProcessPattern>
     max, conformdegree: Integer init 0
// chercher des patrons convenables
     convenientp:= self.ProcessPatternSearching(p)
   //caculer le degré de conformité pour chaque élément de l'ensemble convenientp
   //et marquer celui ayant le degrée le plus élévé
      convenientp.each{ p |conformdegree = Conformity(p.context,c)
                  if (max < conformdegree) then
                        max:=conformdegree
                        result := p   }
end
}
```

Opérateurs d'imitation

Les opérateurs d'imitation sont définis sous forme de méta-opérations des méta-classes *ProcessPatternBinding* et *ProcessPatternApplying* (Figure B et Figure C).

- **ProcessPatternBinding**

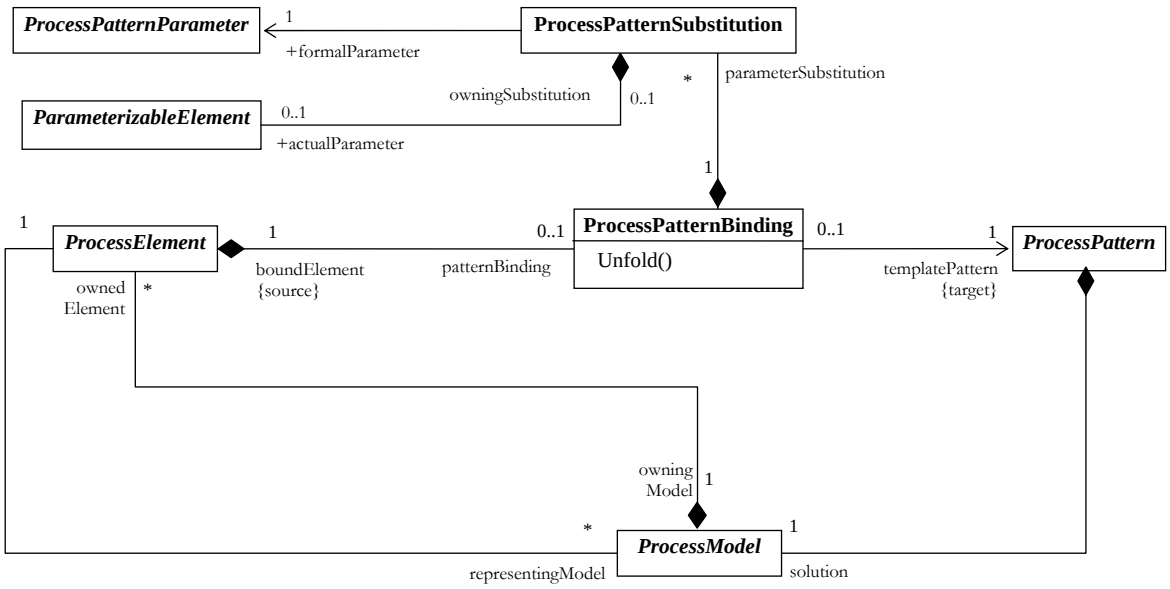


Figure B. Extrait du méta-modèle UML-PP décrivant la relation *ProcessPatternBinding*

Méta-Opération 3. ProcessPatternBinding.Unfold()

```
class ProcessPatternBinding
{
operation Unfold():ProcessModel
is do
var
   // dupliquer la solution du patron template
result:= ProcessPattern.clone(self.templatePattern.solution)
   //substituer les paramètres s'ils sont spécifiés dans la relation
if (self.parameterSubstitution.size()>0)then
    //pour chaque paire de paramètres
self.parameterSubstitution.each{sub|
   //vérifier l'existence d'un paramètre effectif
   if not self.boundElement.owningModel.exists(e|e==sub.actualParameter)
   then  //le paramètre effectif n'existe pas dehors la substitution, il est en fait un
         // juste déclaration du paramètre effectif sous forme StringExpression si le
         // boundElement est au même niveau d'abstraction que le templatePattern
    if (self.boundElement.getAbstractionLevel()==
                          self.templatePattern.getAbstractionLevel())
```

```
    then  //renommer simplement le paramètre formel
      result.each{e|if (e==sub.formalparameter)then
                        ElementRenaming(e,sub.actualParameter)}
       else  // le niveau d'abstraction du boundElement est inférieur à celui du
             // templatePattern, il faut substituer et raffiner le paramètre formel
      result.each{e|if (e==sub.formalparameter)then
          do    // générer un élément ayant le même type mais avec un niveau
                // d'abstraction inférieur à celui du paramètre formel
          var abslevel,type :String
              realactualp:ProcessElement
          abslevel:=self.boundElement.getAbstractionLevel()
          type:=sub.formalparameter.oclIsTypeOf()
            if (type=="Task") then
                if (abslevel=="General") then
                   realactualp:=GeneralTask.new()
                else  realactualp:=ConcreteTask.new()
            else
                if (type=="Product") then
                    if (abslevel=="General") then
                         realactualp:=GeneralProduct.new()
                     else  realactualp:=ConcreteProduct.new()
                else  // type == Role
                    if (abslevel=="General") then
                         realactualp:=GeneralRole.new()
                     else  realactualp:=ConcreteRole.new()
              // le renommer avec le nom spécifié par le paramètre effectif
              realactualp.name:=self.parameterSubstitution.actualParameter)
              // substituer le paramètre formel par l'élément plus spécifique
               ElementRefining(e,realactualp)}
          end
  else  //le paramètre effectif existe, alors substituer le paramètre formel par une copie
        // du paramètre effectif
      result.each{e|if (e==sub.formalparameter) then
      var refactualp :ProcessPattern
          refactualp:=self.boundElement.owningModel.detect(e|
                                                  e==sub.actualParameter)
          ElementSubstituing(e,refactualp)}
        //établir une relation entre l'élément à définir et le modèle élaboré
      self.boundElement.representingModel.add(result)
 end
}
```

- **ProcessPatternApplying**

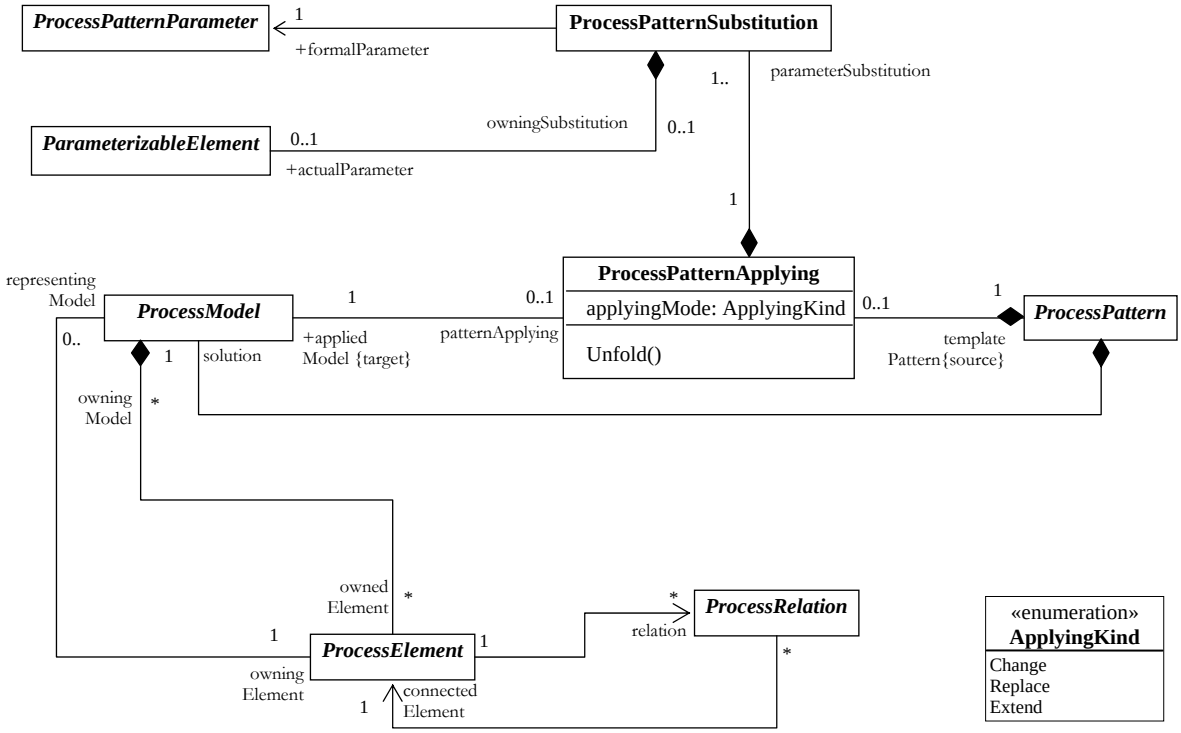


Figure C. Extrait du méta-modèle UML-PP décrivant la relation *ProcessPatternApplying*

Méta-Opération 4. ProcessPatternApplying.Unfold()

```
class ProcessPatternApplying
{
operation Unfold():ProcessModel
is do
var
   // dupliquer la solution du patron template
result:= ProcessPattern.clone(self.templatePattern.solution)
   // substituer les éléments de result correspondant aux paramètres formels
if (self.parameterSubstitution.size()>0)then
// pour chaque paire de paramètres, substituer le paramètre formel par son paramètre effectif
self.parameterSubstitution.each{sub|
  ElementSubstituing(result.detect(e==sub.formalparameter),sub.actualparameter)}
    //si le mode d'application est Change, le processus termine
   // sinon, continuer la fusion du modèle originel pm et du modèle du patron result
if (self.applyingMode!="Change") then
do
var difference : Set<ProcessElement>
    //ajouter à result les éléments du pm qui ne sont pas des paramètres effectifs
```

```
difference := self.appliedModel.reject(e|
                self.parameterSubstitution.actualparameter.exists(f|f==e))
difference.each{e|result.add(e)}
   //dans le modèle result, rétablir les relations entre des éléments originels du appliedModel
self.appliedModel.ownedElement.each{e|
   do
        var erelations: Set<ProcessRelation>
      //identifier l'ensemble de relations auxquelles e participe
         erelations:=e.relation
     //pour chaque relation dans cet ensemble, le recréer dans le modèle résultant si possible
         eraltions.each{r|
                do
                 var f,eres,fres :ProcessElement
                 f :=r.connectedElement
                 eres:=result.detect(x|x==e)
                 fres:=result.detect(z|z==f)
                  //s'il n'y  pas une même relation que r entre eres et fres
                  if not eres.relation.exists(re|(re==r)and
                                    re.connectedElement.exists(fres)) then
                        RelationAdding(eres,fres,r,result)
                   else    //il peut y avoir des conflits
                    //conflit direct
                      if fres.relation.exists(re|(re==r)and
                                    re.connectedElement.exists(eres))then
                      do
                    //remplacer la relation dans result par r seulement si le mode d'application est « Extend »
                          if self.applyingMode=="Extend" then
                            do
                               RelationDeleting(eres,fres,re,result)
                               RelationAdding(eres,fres,r,result)
                             end
                       end
                      else       //conflit indirect
                              ConflitsResolving()
                    RelationDeleting(e,f,r,self.appliedModel)
                    }
     }
     end
   //remplacer le modèle originel par le modèle résultant
self.appliedModel.owningElement.representingModel:=result
end
}
```

ANNEXE B.

DESCRIPTION DU PROCÉDÉ VUP (VIEW-BASED UNIFIED PROCESS)

Cette annexe fournit la description informelle du procédé VUP ainsi que son modèle en UML-PP.

Description informelle du VUP

Le noyau décrit ci-dessous comprend l'analyse et la conception selon la méthode VUML. On laissera de côté l'étape de fin de Revue de conception. Les acteurs intervenant dans ce procédé sont donc les analystes et les concepteurs, l'un d'eux jouant le rôle de chef concepteur.

La première étape, réalisée par les analystes, consiste à analyser le système, i.e. à élaborer le modèle des cas d'utilisation. Cette étape commence par une étude préliminaire du système à développer qui comprend l'analyse de l'existant, la modélisation des aspects métier (éventuellement) et l'établissement d'un glossaire des termes qui seront employés dans la suite du projet. Après cette étude préliminaire, l'analyste construit les cas d'utilisation par l'intermédiaire de 4 tâches. La première tâche consiste à identifier les acteurs du système (qui interagissent avec ce système) et donc les points de vue associés. Pour cela, il faut identifier les types d'acteurs (utilisateurs finaux, développeurs, mainteniciens), structurer les acteurs via le mécanisme de spécialisation/généralisation, et en déduire les points de vue. La seconde tâche permet de lister les besoins des acteurs à partir du cahier des charges, i.e. pour chaque acteur on identifie des activités propres et sa participation aux activités partagées. La troisième tâche consiste à identifier et structurer les cas d'utilisation (comme en UML) en faisant apparaître la participation des acteurs. On regroupe si nécessaire les C.U en paquetages. La quatrième tâche a pour rôle de faire le diagramme des C.U proprement dit, à savoir dessiner les acteurs et les C.U et placer les différentes sortes de flèches reliant ces entités : utilisation, extension, spécialisation.

La deuxième étape, effectuée par les concepteurs, consiste à établir les modèles de conception selon chaque point de vue. Pour chaque acteur (donc point de vue), il s'agit de modéliser en détail les cas d'utilisation dont il est acteur principal, puis de réaliser un diagramme de classes associé au point de vue. La modélisation des cas d'utilisation se fait par l'intermédiaire de 3 tâches séquentielles pour chaque c.u. La première tâche consiste à faire une description textuelle du C.U de la manière suivante : nommer le C.U, identifier les acteurs secondaires, spécifier les préconditions et postconditions éventuelles, identifier le scénario

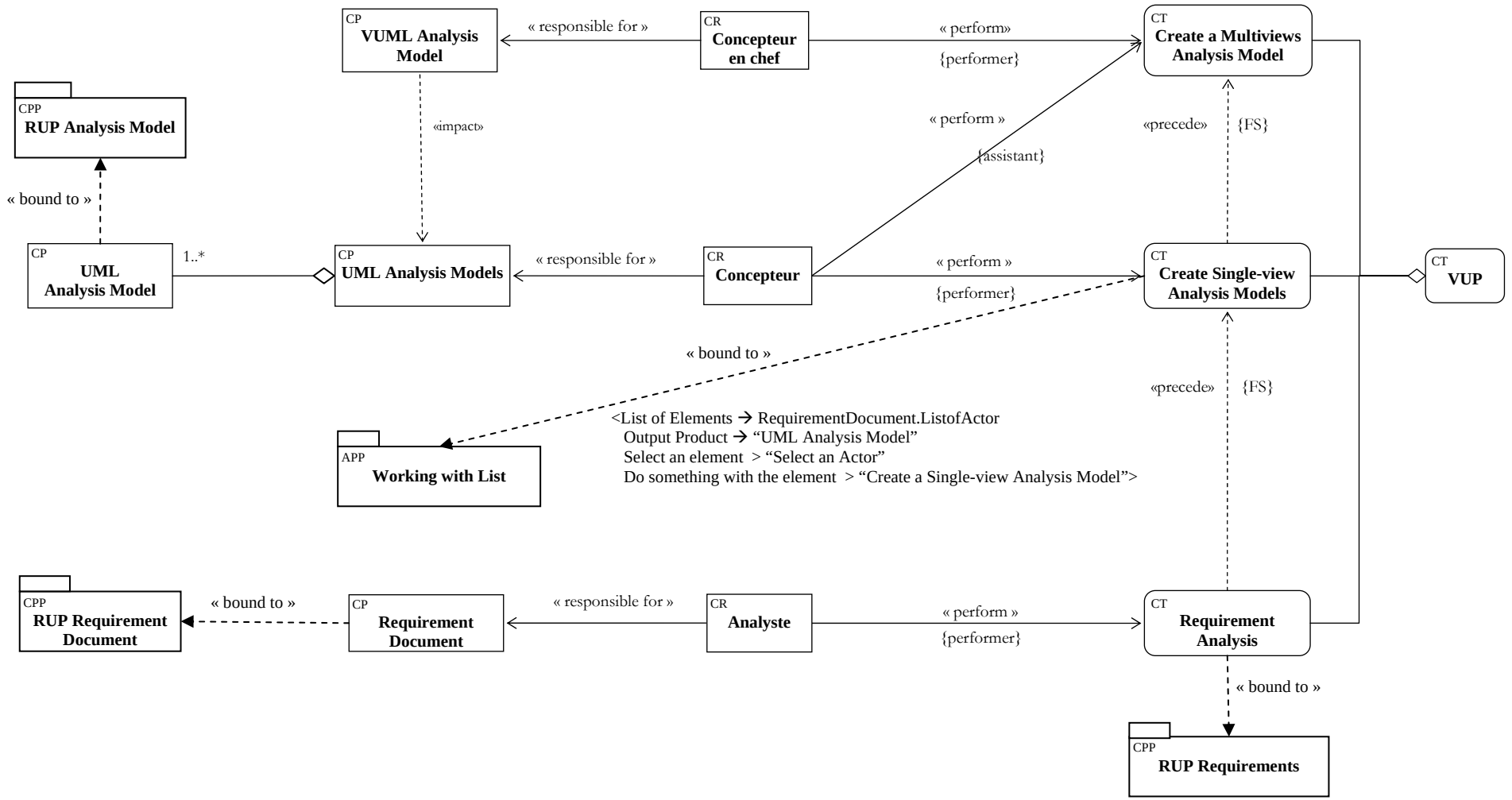
principal, identifier les autres scénarios éventuels. La seconde tâche a pour rôle de traiter les scénarios de façon itérative. Pour chaque scénario, il s'agit d'en faire le résumé textuel, de réaliser le diagramme de contexte (abstrait), d'identifier les objets participant au scénario, de réaliser le diagramme de séquence (détaillé). La troisième tâche a pour objectif la réalisation d'un diagramme de classe préliminaire pour le C.U (en exploitant les diagrammes de séquence de la tâche précédente). Cette tâche se fait via les activités suivantes : identifier et placer les classes du diagramme, spécifier la classe racine du système, dessiner les relations entre les classes. Pour dessiner ces relations, il faut itérer sur la liste des classes en commençant par la racine. Ainsi, pour chaque couple de classes dépendantes choisi, il faut élaborer la relation qui les relie en procédant comme suit : tracer la relation (association, agrégation, composition, …), nommer cette relation, définir un stéréotype éventuel, ajouter les rôles éventuellement, spécifier les cardinalités éventuelles. Il faut ensuite réaliser un diagramme de classes pour le point de vue courant par fusion des diagrammes de classes des différents C.U. Ceci se fait par l'intermédiaire des 3 activités suivantes : classer les C.U du point de vue courant, choisir un diagramme pour initialiser la fusion, réaliser le diagramme de classes par enrichissement progressif du noyau.

La troisième étape, réalisée par le concepteur en chef, consiste à produire le modèle de conception multivues (dite encore fusion) à partir des modèles produits précédemment. Cette étape se compose de 2 tâches. La première a pour but de traiter les incohérences entre modèles monovue. Cette tâche consiste tout d'abord à analyser les diagrammes pour identifier les conflits ou incohérences, puis à élaborer le diagramme de classe multivues VUML sous la forme de 3 activités. La première activité a pour but d'identifier les classes multivues (classes appartenant à au moins 2 modèles monovue). La deuxième activité consiste à modéliser les classes multivues selon un processus itératif : pour chaque classe multivues, il s'agit d'identifier la base de la classe (attributs, méthodes, relations), ajouter les extensions correspondant aux vues, traiter les héritages de vues et les redéfinitions, spécifier les relations de cohérence entre vues ; cette spécification se fait en identifiant les liens de cohérence entre les vues de la classe, formalisant ces liens sous forme de relations de dépendance UML, spécifiant les relations en OCL. La troisième activité a pour but de dessiner le diagramme de classes multivues. Plus précisément, il s'agit de regrouper le cas échéant les classes en paquetages, dessiner les classes et les paquetages (en choisissant le mode de représentation – iconifié ou éclaté – pour les classes multivues), dessiner les relations, mettre les cardinalités.

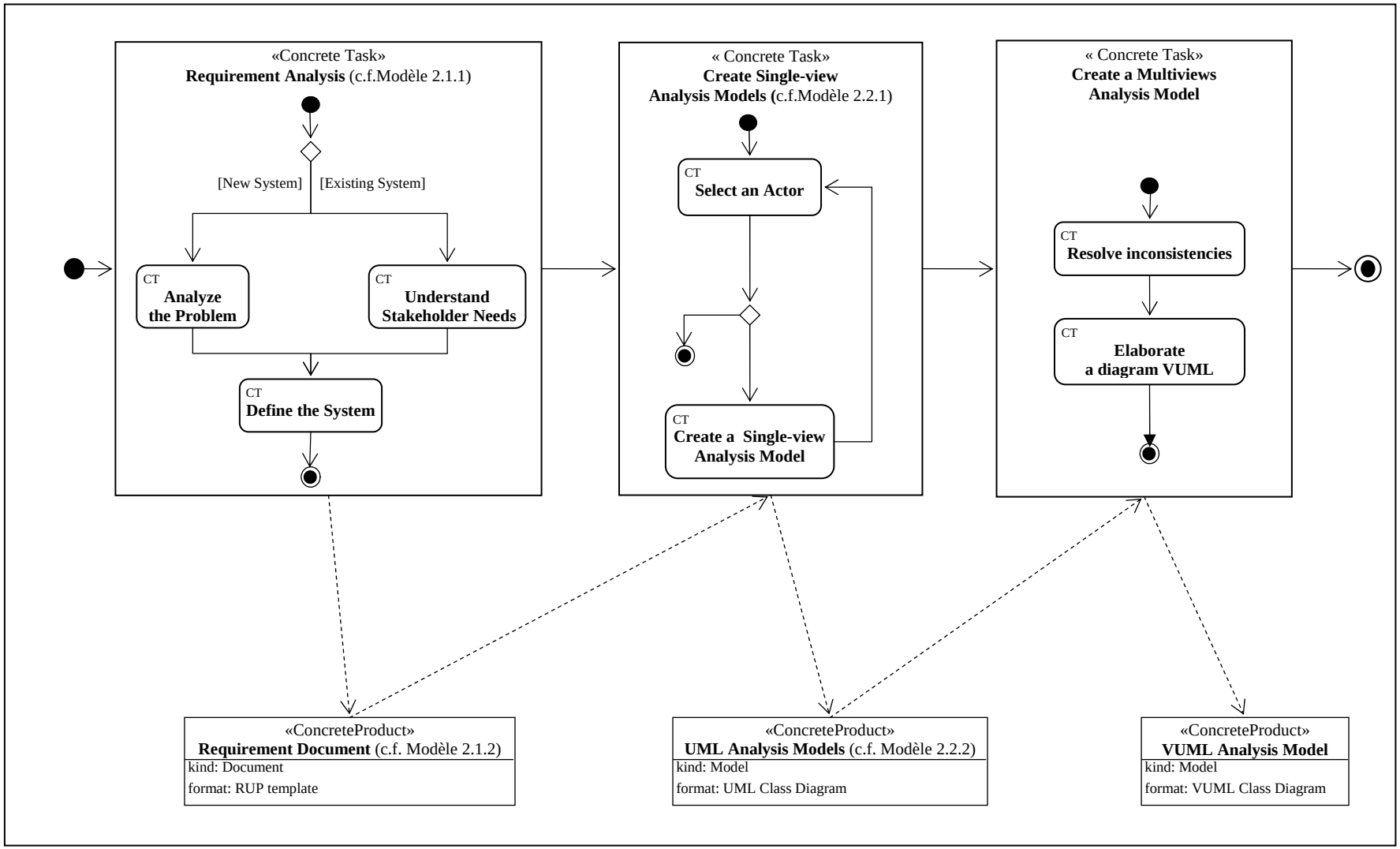
Modèle du VUP en UML-PP

Dans cette section nous présentons quelques modèles en UML-PP des éléments principaux du procédé VUP.

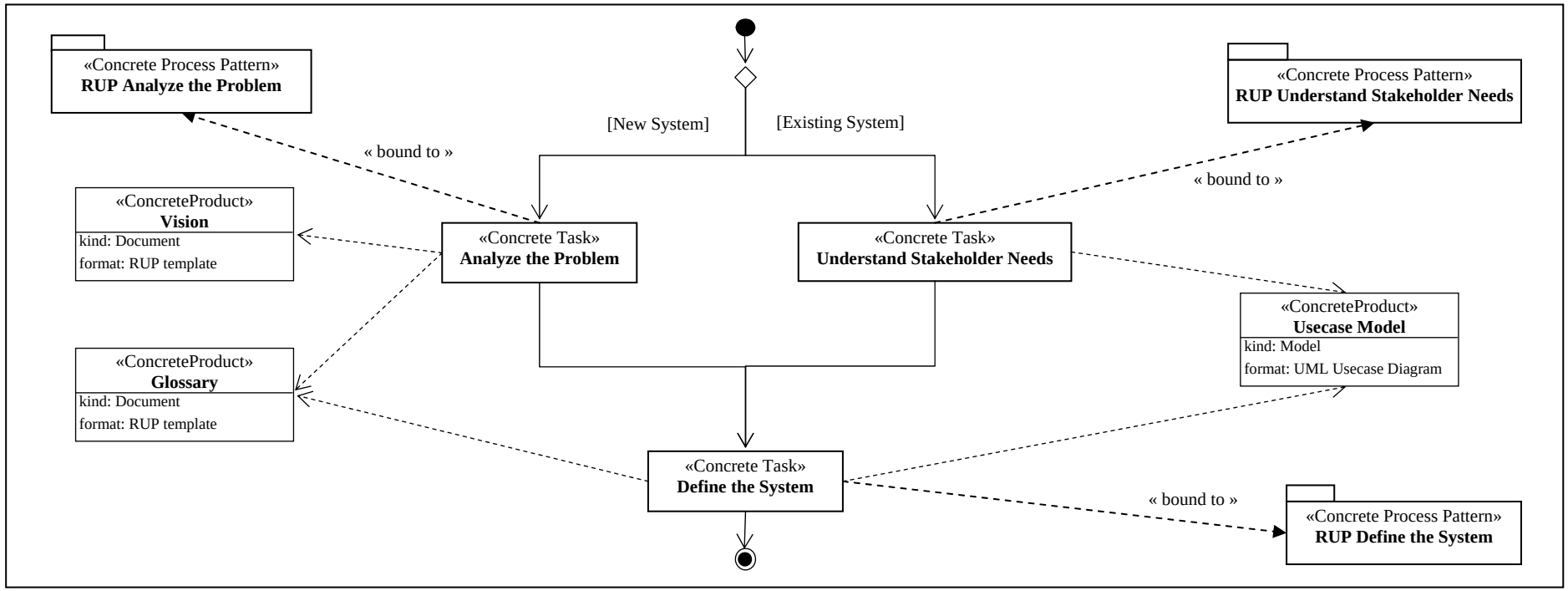
1. Structure statique du procédé VUP



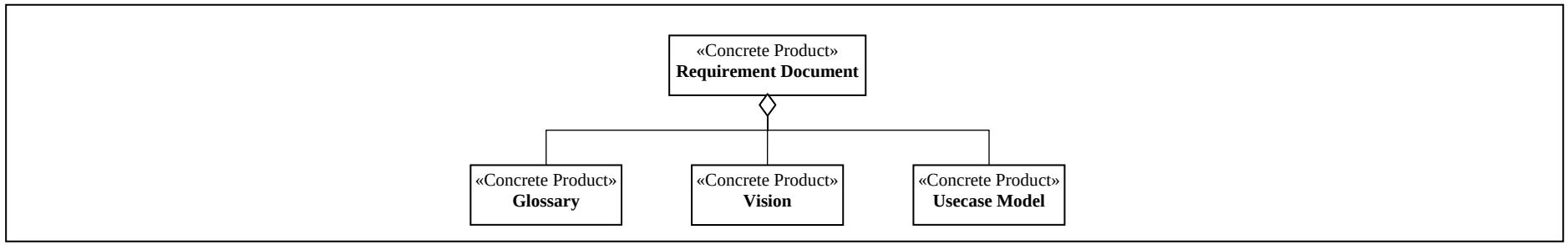
2. Comportement dynamique du procédé VUP



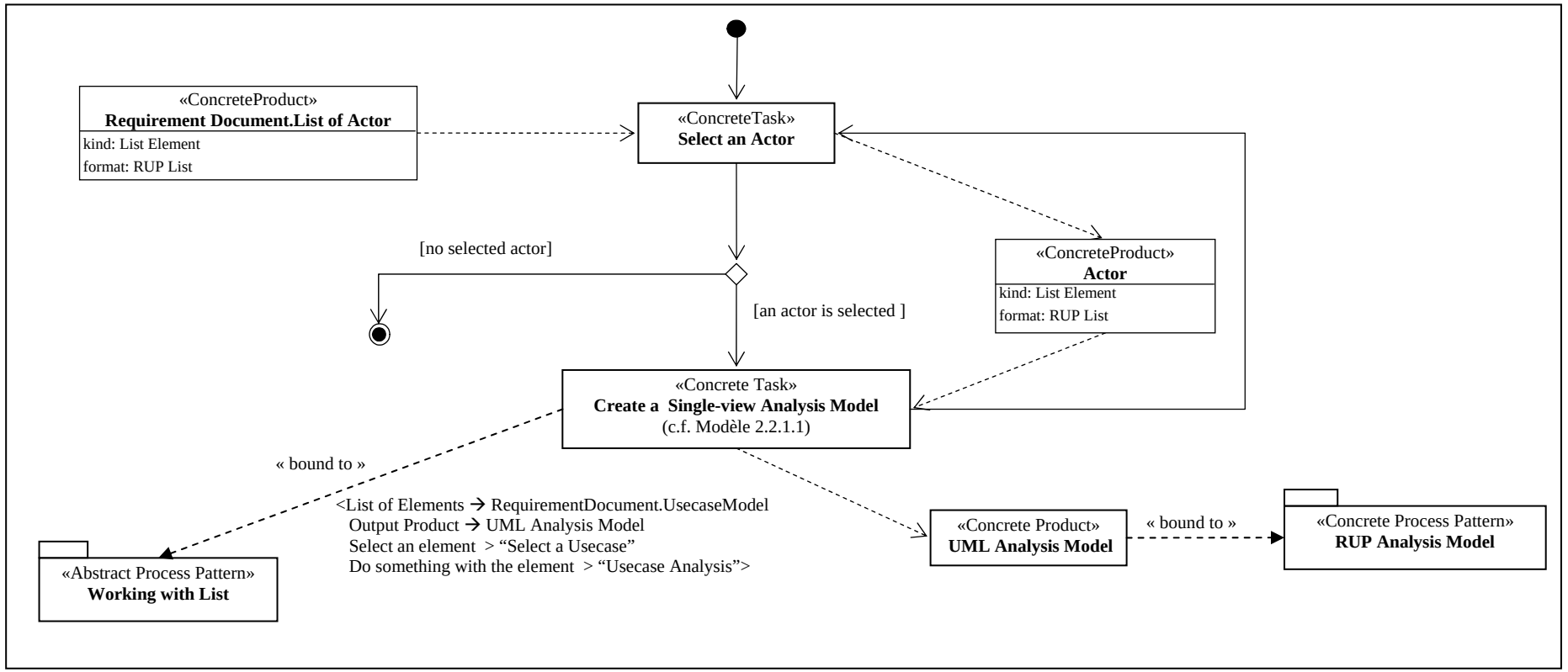
2.1.1. Modèle décrivant la tâche VUP - Requirement Analysis



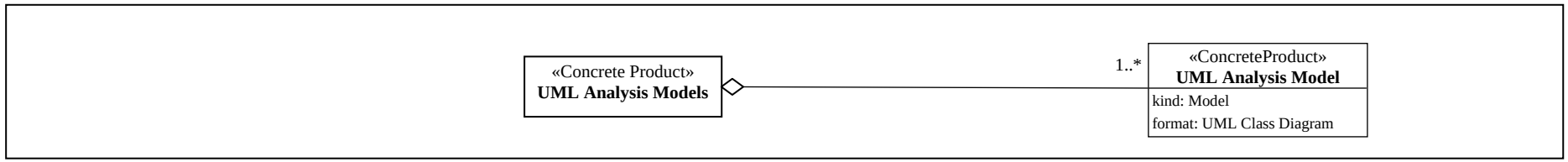
2.1.2. Modèle décrivant le produit VUP - Requirement Document



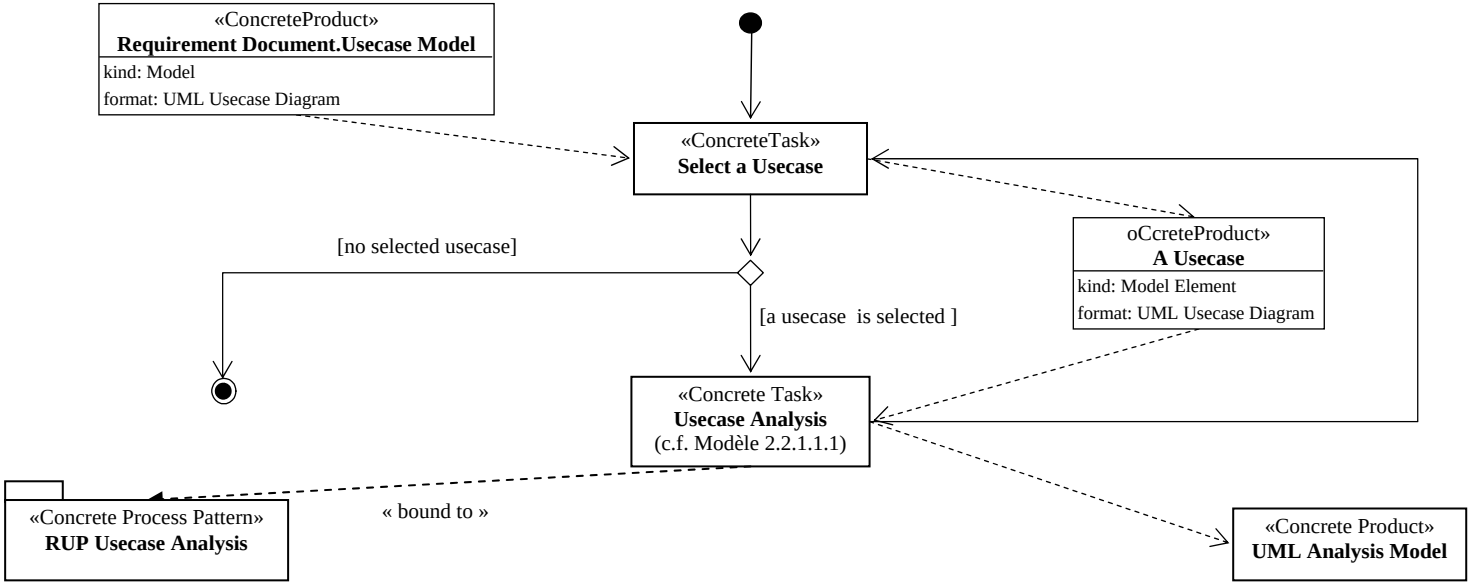
2.2.1. Modèle décrivant la tâche VUP- Create Single-view Analysis Models



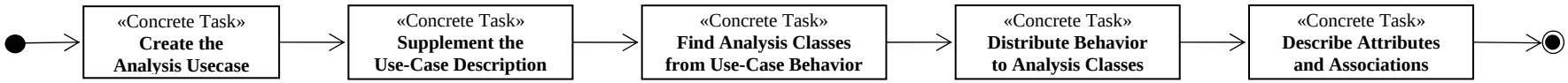
2.2.2. Modèle décrivant le produit VUP- UML Analysis Models



2.2.1.1. Modèle décrivant la tâche VUP- Create a Single-view Analysis Model



2.2.1.1.1. Modèle décrivant la tâche VUP- Usecase Analysis



ANNEXE C.

SOURCE C# DE PATPRO-MOD

Cette annexe fournit un extrait du code source C# de l'environnementPATPRO-MOD.

Organisation du code source

Le code source du PATPRO-MOD est organisé en deux paquetages principaux : le paqutage *Entity-Classes* contient des classes définissant les concepts du UML-PP, et le paquetage *Drawing-Classes* contient ceux permettant de dessiner des modèles UML-PP (Figure D).

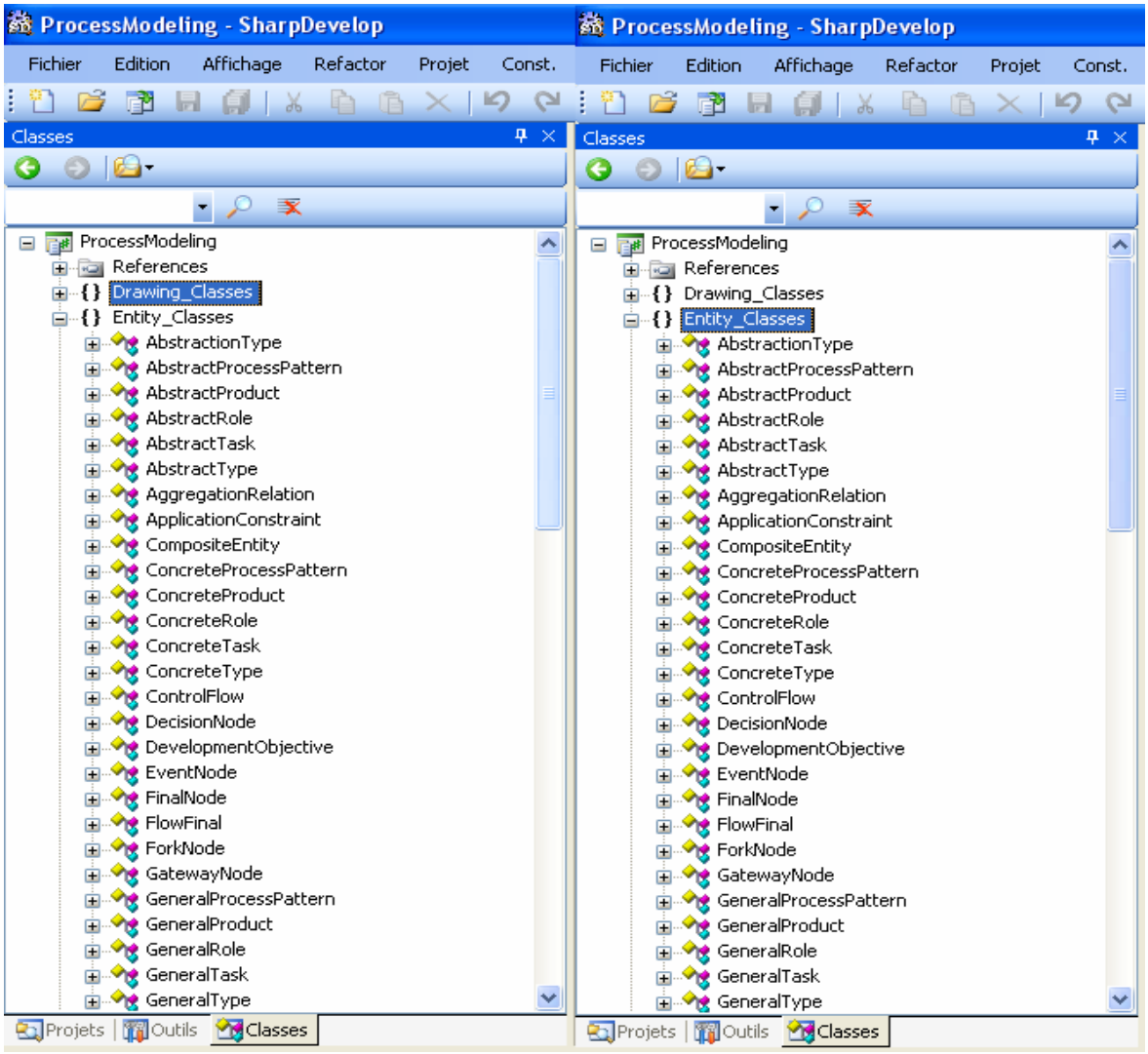


Figure D. Organisation du code source PATPRO-MOD

Dans la suite nous extrayons le code des classes concernant l'organisation et la modélisation de patrons de procédé.

Classe ProcessPattern

```
// définir le concept ProcessPattern de UML-PP
using System;
using System.Collections.Generic;
using System.Text;
using ProcessModeling.Entity_Classes;
using ProcessModeling.Drawing_Classes;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
namespace ProcessModeling.Entity_Classes
{
    [Serializable()] public class ProcessPattern:ProcessElement
    {
        private DiagramDrawing m_DiagramDrawing;
        public virtual string GetAbstractionTypeShortString()
        {            return "";
        }
        public DiagramDrawing SolutionDrawing
        {
            get { return m_DiagramDrawing; }
            set { m_DiagramDrawing = value; }
        }
        private string signature;
        public string Signature
        {
            get { return signature; }
            set { signature = value; }
        }
        public ProcessPattern()
        {
            PatternProblem = new PatternProblem();
            PatternContext = new PatternContext();
        }
        private PatternProblem m_problem;
        public PatternProblem PatternProblem
        {
            get { return m_problem; }
            set { m_problem = value; }
        }
        private PatternContext m_context;
        public PatternContext PatternContext
        {
            get { return m_context; }
            set { m_context = value; }
        }
public static void SaveProcessPatternToFile(ProcessPattern d, string fileName)
        {
            FileStream stream = File.OpenWrite(fileName);
            BinaryFormatter f = new BinaryFormatter();
            f.Serialize(stream, d);
            stream.Close();
        }
public static ProcessPattern GetProcessPatternFromFile(string fileName)
        {
            FileStream stream = File.OpenRead(fileName);
            BinaryFormatter f = new BinaryFormatter();
            ProcessPattern d = (ProcessPattern)f.Deserialize(stream);
            return d;
        }
    }
}
```

Classe PatternProblem

```
// définir le concept PatternProblem de UML-PP

using System;
using System.Collections.Generic;
using System.Text;

namespace ProcessModeling.Entity_Classes
{
    [Serializable()] public class PatternProblem : CompositeEntity
    {

    }
}
```

Classe PatternContext

```
/// définir le concept PatternContext de UML-PP

using System;
using System.Collections.Generic;
using System.Text;

namespace ProcessModeling.Entity_Classes
{
    [Serializable()] public class PatternContext
    {
        public PatternContext()
        {
            ResultingContext = new ApplicationConstraint();
            InitialContext = new ApplicationConstraint();
            ReuseSituation = new ReuseSituation();
        }
        private ReuseSituation reuseSituation;
        public ReuseSituation ReuseSituation
        {
            get { return reuseSituation; }
            set { reuseSituation = value; }
        }
         private ApplicationConstraint initialContext;
```

```
        public ApplicationConstraint InitialContext
        {
            get { return initialContext; }
            set { initialContext = value; }
        }
        private ApplicationConstraint resultingContext;
        public ApplicationConstraint ResultingContext
        {
            get { return resultingContext; }
            set { resultingContext = value; }
        }

    }
}
```

Classe BehaviorDiagramDrawing (extrait)

```
/// dessiner un diagramme d'activité qui représente un modèle
/// comportemental de procédé, et qui peut être capturé comme
/// solution d'un patron

using System;
using System.Collections.Generic;
using System.Text;
using System.Drawing;
using ProcessModeling.Utility_Classes;
using ProcessModeling.Entity_Classes;
namespace ProcessModeling.Drawing_Classes
{
    [Serializable()] public class BehaviorDiagramDrawing:DiagramDrawing
    {
        private ProductDrawingList m_ProductDrawingList;
        private TaskDrawingList m_TaskDrawingList;
        private RelationDrawingList m_RelationList;
        private StepDrawingList m_StepDrawingList;
        private EventDrawingList m_EventDrawingList;
        private GatewayNodeDrawingList m_GatewayNodeDrawingList;
        private SynchronisationNodeDrawingList m_SynchronisationNodeDrawingList;
        private ProcessPatternDrawingList m_ProcessPatternDrawingList;

        public ProcessPatternDrawingList ProcessPatternDrawings
        {
            get { return m_ProcessPatternDrawingList; }
            set { m_ProcessPatternDrawingList = value; }
        }
        public BehaviorDiagramDrawing()
        {
            ProductDrawings = new ProductDrawingList();
            TaskDrawings = new TaskDrawingList();
            RelationDrawings = new RelationDrawingList();
            StepDrawings = new StepDrawingList();
            EventNodeDrawings = new EventDrawingList();
            GatewayNodeDrawings = new GatewayNodeDrawingList();
            SynchronisationNodeDrawings = new SynchronisationNodeDrawingList();
            ProcessPatternDrawings = new ProcessPatternDrawingList();
        }
// obtenir les éléments à dessiner

    public override ElementDrawingList GetProcessElementDrawings()
        {
            ElementDrawingList list = new ElementDrawingList();
            foreach (ProductDrawing product in ProductDrawings)
                list.Add(product);
            foreach (TaskDrawing product in TaskDrawings)
                list.Add(product);
            foreach (StepDrawing product in StepDrawings)
                list.Add(product);
            return list;
        }
        public SynchronisationNodeDrawingList SynchronisationNodeDrawings
        {
            …
        }
        public GatewayNodeDrawingList GatewayNodeDrawings
        {
            …
        }
        public EventDrawingList EventNodeDrawings
        {
            …
        }
        public StepDrawingList StepDrawings
        {
            …
        }
        public RelationDrawingList RelationDrawings
        {
            …
        }
        public ProductDrawingList ProductDrawings
        {
            …

        }
        public TaskDrawingList TaskDrawings
        {
            …
        }

       // dessiner les éléments d'un diagramme

        public override void DrawOn(Graphics g)
        {
            foreach (ProductDrawing product in ProductDrawings)
                product.DrawOn(g);
            foreach (TaskDrawing task in TaskDrawings)
```

```
            task.DrawOn(g);
        foreach (StepDrawing step in StepDrawings)
            step.DrawOn(g);
        foreach (RelationDrawing relation in RelationDrawings)
            relation.DrawOn(g);
        foreach (EventNodeDrawing node in EventNodeDrawings)
            node.DrawOn(g);
        foreach (GatewayNodeDrawing node in GatewayNodeDrawings)
            node.DrawOn(g);
        foreach (SynchronisationNodeDrawing node in
                 SynchronisationNodeDrawings)
            node.DrawOn(g);
        foreach (ProcessPatternDrawing p in ProcessPatternDrawings)
            p.DrawOn(g);
    }
  public override EntityDrawing GetEntityDrawingFromPoint(int x, int y)
    {  …
    }
 public override ElementDrawing GetElementDrawingFromPoint(int x, int y)
    { …
    }
 public void AddProductDrawing(ProductDrawing productDrawing)
    {
        ProductDrawings.Add(productDrawing);
    }
 public void AddTaskDrawing(TaskDrawing taskDrawing)
    {
        TaskDrawings.Add(taskDrawing);
    }

// dessiner une relation de procédé ou une relation d'application
// de patrons de procédé

public void AddRelationDrawing(RelationDrawing relationDrawing)
{

 RelationDrawings.Add(relationDrawing);

//si la relation est «binding», appeler la fonction
//ProcessPatternBindingDrawing.UnFoldHandler pour la déplier

if (relationDrawing is ProcessPatternBindingDrawing)
 ((ProcessPatternBindingDrawing)relationDrawing).UnFoldEvent +=new
ProcessPatternBindingDrawing.UnFoldHandler(BehaviorDiagramDrawing_UnFoldEvent);

//si la relation est «applying», appeler la fonction
//ProcessPatternApplyingDrawing.UnFoldHandler pour la déplier

if (relationDrawing is ProcessPatternApplyingDrawing)
((ProcessPatternApplyingDrawing)relationDrawing).UnFoldEvent += new
ProcessPatternApplyingDrawing.UnFoldHandler(BehaviorDiagramDrawing_UnFoldEvent);

}
```

```
//déplier une relation ProcessPatternApplying

void BehaviorDiagramDrawing_UnFoldEvent(ProcessPatternApplyingDrawing e)
  {  try
            {
            ProcessPatternDrawing patternDrawing =
                    (ProcessPatternDrawing)e.TargetElement;
            ProcessPattern processPattern =
                    (ProcessPattern)patternDrawing.GetEntity();
/Xoa di ky hieu process pattern tuong ung trong mo hinh
           this.ProcessPatternDrawings.Remove(patternDrawing);

//Them cac phan tu cua process pattern vao trong mo hinh
      foreach (ElementDrawing ele in
          processPattern.SolutionDrawing.GetUnfoldElementDrawingItems())
                {
                    if (ele is ProductDrawing)
                    {
                        this.ProductDrawings.Add(ele);
                        ((Product)ele.GetEntity()).AbstractionType =
                               new ConcreteType();
                    }
                    if (ele is TaskDrawing)
                    {
                        this.TaskDrawings.Add(ele);
                        ((Task)ele.GetEntity()).AbstractionType =
                               new ConcreteType();
                    }
                    if (ele is StepDrawing)
                        this.StepDrawings.Add(ele);
                    if (ele is GatewayNodeDrawing)
                        this.GatewayNodeDrawings.Add(ele);
                    if (ele is SynchronisationNodeDrawing)
                        this.SynchronisationNodeDrawings.Add(ele);
                    if (ele is EventNodeDrawing)
                        this.EventNodeDrawings.Add(ele);
                }
  //Them cac relation tuong ung vao mo hinh
      foreach (RelationDrawing rel in
            processPattern.SolutionDrawing.GetUnfoldRelationDrawingItems())
                    this.RelationDrawings.Add(rel);
   //Xoa di applying relation dang xet
      this.RelationDrawings.Remove(e);
    //Dieu chinh cac entity moi duoc them vao
            foreach (ApplyingParameter para in e.ApplyingParameterList)
                {
                   para.SourceElementDrawing.GetEntity().Name =
                      para.TargetElementDrawing.GetEntity().Name;
                }
            ElementDrawingList deletedElement = new ElementDrawingList();
            ElementDrawingList replacedElement = new ElementDrawingList();

//Xoa bo cac entity hien co
            foreach (ApplyingParameter para in e.ApplyingParameterList)
```

```
            {
                ElementDrawing ele = para.TargetElementDrawing;
                deletedElement.Add(ele);
                replacedElement.Add(para.SourceElementDrawing);
                if (ele is ProductDrawing)
                {
                    this.ProductDrawings.Remove(ele);
                }
                if (ele is TaskDrawing)
                {
                   this.TaskDrawings.Remove(ele);
                }
                if (ele is StepDrawing)
                    this.StepDrawings.Remove(ele);
                if (ele is GatewayNodeDrawing)
                    this.GatewayNodeDrawings.Remove(ele);
                if (ele is SynchronisationNodeDrawing)
                    this.SynchronisationNodeDrawings.Remove(ele);
                if (ele is EventNodeDrawing)
                    this.EventNodeDrawings.Remove(ele);
            }
//Xoa va hieu chinh cac relation co lien quan den cac entity bi xoa
            for (int i=0;i<RelationDrawings.Count;i++)
            {
                RelationDrawing rel = RelationDrawings[i];
                bool source = false;
                bool target = false;
                source = deletedElement.Contains(rel.SourceElement);
                target = deletedElement.Contains(rel.TargetElement);
                if (source == true && target == true)
                {
                    RelationDrawings.Remove(rel);
                    i--;
                }
                else if (source == true && target == false)
                {
                    int index = deletedElement.IndexOf(rel.SourceElement);
                    rel.SourceElement = replacedElement[index];
                }
                else if (source == false && target == true)
                {
                    int index = deletedElement.IndexOf(rel.TargetElement);
                    rel.TargetElement = replacedElement[index];
                }
            }
            this.RaiseReDrawEvent();
        }
        catch
        {

        }
    }
```

```
//déplier une relation ProcessPatternBinding
void BehaviorDiagramDrawing_UnFoldEvent(ProcessPatternBindingDrawing e)
       {
          try
           {
          ProcessPatternDrawing patternDrawing =
                (ProcessPatternDrawing)e.TargetElement;
          ProcessPattern processPattern =
                (ProcessPattern)patternDrawing.GetEntity();

//Xoa di ky hieu process pattern tuong ung trong mo hinh
     this.ProcessPatternDrawings.Remove(patternDrawing);

//Them cac phan tu cua process pattern vao trong mo hinh
    foreach (ElementDrawing ele in
processPattern.SolutionDrawing.GetUnfoldElementDrawingItems())
    {
       if (ele is ProductDrawing)
          {
            this.ProductDrawings.Add(ele);
            ((Product)ele.GetEntity()).AbstractionType = new ConcreteType();
                  }
                  if (ele is TaskDrawing)
                  {
                      this.TaskDrawings.Add(ele);
                ((Task)ele.GetEntity()).AbstractionType = new ConcreteType();
                  }
                  if (ele is StepDrawing)
                      this.StepDrawings.Add(ele);
                  if (ele is GatewayNodeDrawing)
                      this.GatewayNodeDrawings.Add(ele);
                  if (ele is SynchronisationNodeDrawing)
                      this.SynchronisationNodeDrawings.Add(ele);
                  if (ele is EventNodeDrawing)
                      this.EventNodeDrawings.Add(ele);
              }
  //Them cac quan he tuong ung tu process pattern vao trong mo hinh
         foreach (RelationDrawing rel in
processPattern.SolutionDrawing.GetUnfoldRelationDrawingItems())
                  this.RelationDrawings.Add(rel);
               //Xoa di quan he binding
               this.RelationDrawings.Remove(e);
               //Dieu chinh cac relation cho thich hop
          ElementDrawing startElement =
             processPattern.SolutionDrawing.GetUnfoldStartDrawingItem();
          ElementDrawing endElement =
              processPattern.SolutionDrawing.GetUnfoldEndDrawingItem();
               //chinh cac relation di toi no
               if (startElement != null)
                   foreach (RelationDrawing relation in this.RelationDrawings)
                       if (relation.TargetElement == e.SourceElement)
                           relation.TargetElement = startElement;
               if (endElement != null)
                   foreach (RelationDrawing relation in this.RelationDrawings)
```

```
                    if (relation.SourceElement == e.SourceElement)
                        relation.SourceElement = endElement;
                //Thay doi cac ten lai cho phu hop
                foreach (BindingParameter para in e.BindingParameterList)
                para.ElementDrawingParameter.GetEntity().Name =
                      para.Value.ToString();
            }
            catch

        }
        public override ElementDrawingList GetUnfoldElementDrawingItems()
        {            …
        }
        public override RelationDrawingList GetUnfoldRelationDrawingItems()
        {          …
        }
        public override ElementDrawing GetUnfoldStartDrawingItem()
        {  …
        }
        public override ElementDrawing GetUnfoldEndDrawingItem()
        {  …
        }
        public void AddStepDrawing(StepDrawing stepDrawing)
        {
            StepDrawings.Add(stepDrawing);
        }
        public override void RemoveEntity(EntityDrawing entity)
        {
            ProductDrawings.Remove(entity);
            TaskDrawings.Remove(entity);

            RelationDrawings.Remove(entity);
        }
        public void AddEventNodeDrawing(EventNodeDrawing node)
        {
            EventNodeDrawings.Add(node);
        }
        public void AddGatewayNodeDrawing(GatewayNodeDrawing node)
        {
            GatewayNodeDrawings.Add(node);
        }
      public void AddSynchronisationNodeDrawing(SynchronisationNodeDrawing node)
        {
            SynchronisationNodeDrawings.Add(node);
        }
        public void AddProcessPatternDrawing(ProcessPatternDrawing node)
        {
            ProcessPatternDrawings.Add(node);
        }
    }
}
```

Classe ProcessPatternBinding

```
// definir la relation ProcessPatternBinding comme
// une relation de réutilisation

using System;
using System.Collections.Generic;
using System.Text;

namespace ProcessModeling.Entity_Classes
{
    [Serializable()]
    public class ProcessPatternBinding : ReuseFlow
    {
    }
}
```

Classe ProcessPatternApplying

```
// definir la relation ProcessPatternApplying comme
// une relation de réutilisation

using System;
using System.Collections.Generic;
using System.Text;


namespace ProcessModeling.Entity_Classes
{
    [Serializable()]
    public class ProcessPatternApplying : ReuseFlow
    {
    }

}
```

Classe ProcessPatternBindingDrawing

```
// dessiner la relation ProcessPatternBinding
// dans un modèle de procédé

using System;
using System.Collections.Generic;
using System.Text;
using ProcessModeling.Drawing_Classes;
using ProcessModeling.Entity_Classes;
using ProcessModeling.Utility_Classes;
namespace ProcessModeling.Drawing_Classes
{
    [Serializable()]
  public class ProcessPatternBindingDrawing : ReuseFlowDrawing
```

```
    {
   public delegate void UnFoldHandler(ProcessPatternBindingDrawing e);
       public event UnFoldHandler UnFoldEvent;
       private BindingParameterList m_ParameterList;
        public BindingParameterList BindingParameterList
        {
            get { return m_ParameterList; }
            set { m_ParameterList = value; }
        }
public void UnFold()
        {
            if (UnFoldEvent != null)
                UnFoldEvent(this);
        }
public override void DrawBody(System.Drawing.Graphics g)
        {
                base.DrawBody(g);
        }
   }
}
```

Classe ProcessPatternApplyingDrawing

```
// dessiner la relation ProcessPatternApplying
// dans un modèle de procédé

using System;
using System.Collections.Generic;
using System.Text;
using ProcessModeling.Drawing_Classes;
using ProcessModeling.Entity_Classes;
using ProcessModeling.Utility_Classes;
namespace ProcessModeling.Drawing_Classes
{
    [Serializable()]
    public class ProcessPatternApplyingDrawing : ReuseFlowDrawing
    {
        public delegate void UnFoldHandler(ProcessPatternApplyingDrawing e);
        public event UnFoldHandler UnFoldEvent;
        private ApplyingParameterList m_ParameterList;
        public ApplyingParameterList ApplyingParameterList
        {
                get { return m_ParameterList; }
            set { m_ParameterList = value; }
        }
        public void UnFold()
        {
            if (UnFoldEvent != null)
                UnFoldEvent(this);
        }
  }
```

Classe FormProcessPatternList

```
// gérer la liste de patrons et permettre de sélectionner un patron

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using ProcessModeling.Entity_Classes;
using ProcessModeling.Drawing_Classes;
using ProcessModeling.Utility_Classes;
namespace ProcessModeling.Forms
{
    public partial class FormProcessPatternList : Form
    {
        public ProcessPattern SelectedProcessPattern
        {
            get
            {
                if (listView1.SelectedItems.Count > 0)
                    return (ProcessPattern)listView1.SelectedItems[0].Tag;
                return null;
            }

        }
        public FormProcessPatternList()
        {
            InitializeComponent();
        }

        private void FormProcessPatternList_Load(object sender, EventArgs e)
        {
            this.Name = "Process Pattern List";
         foreach (ProcessPattern pattern in GlobalMemberList.ProcessPatterns)
            {
                ListViewItem item = new ListViewItem(pattern.Name);
                item.SubItems.Add(pattern.GetAbstractionTypeShortString());
                item.SubItems.Add(pattern.PatternProblem.Name);
                item.SubItems.Add("Initial Context:" +
pattern.PatternContext.InitialContext.Name + "\r\nResulting Context:" +
pattern.PatternContext.ResultingContext.Name);
                item.Tag = pattern;
                listView1.Items.Add(item);
            }
        }
    }
}
```

MODÉLISATION DE PROCÉDÉS LOGICIELS
À BASE DE PATRONS RÉUTILISABLES

Résumé :

Cette thèse est consacrée à la réutilisation de procédés par une approche à base de patrons de procédé. Le concept de patron de procédé a été introduit pour capitaliser et réutiliser des solutions éprouvées des problèmes récurrents liés à la modélisation de procédés logiciels. Cependant cette approche est encore peu exploitée à cause du champ de définition limité, du manque de formalisation, de méthodologie et d'outils support. Pour promouvoir l'utilisation de patrons de procédé et réduire l'effort de modélisation, nous considérons le concept de patron de procédé à différents niveaux d'abstraction pour capturer divers types de connaissances sur les procédés, et proposons des moyens pour réutiliser de façon (semi-)automatique ces patrons dans la modélisation de procédés.

Nous avons défini le méta-modèle de procédé UML-PP pour formaliser le concept de patron de procédé et la manière d'appliquer les patrons dans la modélisation de procédés. UML-PP permet de décrire la structure interne d'un patron de procédé ainsi que les relations entre patrons, et permet d'exprimer explicitement l'utilisation de patrons dans les modèles de procédé. Nous proposons le méta-procédé PATPRO définissant une démarche de modélisation pour élaborer un modèle de procédé UML-PP en réutilisant des patrons de procédé. Pour permettre une automatisation de l'application de patrons de procédé, nous définissons une sémantique opérationnelle des opérateurs de réutilisation de patrons qui réalisent l'imitation de patrons. Nous avons réalisé le prototype PATPRO-MOD permettant de gérer des catalogues de patrons de procédé et d'élaborer des modèles de procédé UML-PP en réutilisant semi-automatiquement des patrons prédéfinis.

Mots-clés : patron de procédé, modélisation de procédé, réutilisation de procédé

SOFTWARE PROCESS MODELLING BASED ON REUSABLE PATTERNS

Abstract :

This thesis investigates the reuse of software processes by an approach based on process patterns. The objective of our work is to make process patterns directly applicable in process modeling. The concept of process pattern is used to capture and reuse the proven solutions for recurring modelling process problems. However, this attractive concept has still been poorly exploited due to the inadequate formalization and the lack of supporting methodology and tools. To promote the use of process patterns and reduce the modelling effort, we broaden the concept of process pattern for capturing various types of process knowledge at different abstract levels, and propose ways to reuse (semi-)automatically process patterns in process modelling.

We define the process meta-model UML-PP to formalize the process pattern concept and the ways to apply patterns in process models. UML-PP allows describing the internal structure of a process pattern as well as the relations between process patterns, and enables the explicit representation of process patterns' applications in process models. We propose the meta-process PATPRO defining the modelling steps to elaborate a process model in UML-PP by reusing process patterns. To allow automated applications of process patterns, we define an operational semantics for the patterns reuse operators who carry out some tasks of the meta-process. We have developed the prototype PATPRO-MOD allowing to manage process patterns catalogues, and to elaborate process models in UML-PP by reusing (semi-)automatically process patterns.

Keywords : process pattern, process modelling, process reuse